

PiDuce – a project for experimenting Web services technologies^{*}

Samuele Carpineti^a, Cosimo Laneve^{a,*}, Luca Padovani^b

^a*University of Bologna, Department of Computer Science,
Mura Anteo Zamboni, 7, 40127 Bologna, Italy*

^b*University of Urbino, Information Science and Technology Institute,
Piazza della Repubblica, 13, 61029 Urbino, Italy*

Abstract

The PiDuce project aims at experimenting Web services technologies by relying on solid theories about process calculi and formal languages for XML documents and schemas.

The PiDuce programming language has values and datatypes that extend XML documents and schemas with channels, it uses a pattern matching mechanism for deconstructing values, and it retains control constructs that are based on Milner's asynchronous π -calculus. The compiler yields typesafe XML object codes by means of a powerful type system with subtyping. The runtime supports the execution of programs over networks by relying on state-of-the-art technologies, such as XML schema and WSDL, thus enabling interoperability with existing Web services.

In this paper we thoroughly describe the PiDuce project: the programming language and its semantics, the architecture of the distributed runtime and its implementation. A fully-functional prototype is available at www.cs.unibo.it/PiDuce/.

Key words: π -calculus, XML schema, type system, subschema relation, WSDL, Web services.

^{*} Aspects of this investigation were supported in part by a Microsoft initiative in concurrent computing and Web services.

^{*} Corresponding author.

Email addresses: carpinet@cs.unibo.it (Samuele Carpineti),
laneve@cs.unibo.it (Cosimo Laneve), padovani@sti.uniurb.it (Luca Padovani).

1 Introduction

The PiDuce project (www.cs.unibo.it/PiDuce) has two main motivations. The first one is to provide a distributed implementation of the asynchronous π -calculus [8]. In asynchronous π -calculus, a program uses channels and message passing to interact with other programs. A natural distributed setting is to let each channel belong to a single location. For instance, there is one location for the channels x, y, z and another for u, v . A *service* $x(w).P$ moves to the first location; it waits to receive the formal parameter w and then continues as P . If a *service request* $\bar{x}[v]$ should arise anywhere else in the system, it knows where to find the matching input resource. This basic model has been extensively studied (the $\pi_{1\ell}$ calculus [3], the local π -calculus [33]), and is used in previous distributed prototypes (the join calculus [16], Microsoft Biztalk Server [25]).

However, in this basic model, one immediately faces the problem of *input capability*, which is the ability in the (asynchronous) π -calculus, to receive a channel name and subsequently accept inputs on it. Consider the example $x(u).u(v).Q$. This program is located at (the location of) x , but upon reaction with $\bar{x}[w]$ it produces the continuation $w(v).Q\{w/u\}$ – and this continuation is still at x , whereas it should actually be at w . Solving the problem of input capability is a key challenge in distributing the π -calculus. The join calculus, the local pi calculus and the $\pi_{1\ell}$ calculus simply disallow input capability, on the grounds that it seems un-implementable. That is, in a term $x(u).P$, the process P may not contain any inputs on channel u . Biztalk offers input capability when run over a reliable message service (MSMQ) but not otherwise [25]. Implementation details of Biztalk have not been published; in this respect this paper may be seen as a formal alternative implementation.

We solve the input capability problem in PiDuce using the theory of linear forwarders [18]. The solution consists of allowing just a limited atom of input capability – the *linear forwarder*. A linear forwarder $x \multimap y$ is a process that simply turns one message on x into a message on y . (A linear forwarder $x \multimap y$ may be safely considered as just the π -calculus process $x(u).\bar{y}[u]$.) To illustrate how linear forwarders enable input capability, consider the process $x(u).u(v).Q$. Then it is encoded as

$$x(u).(u')(u \multimap u' \mid u'(v).Q)$$

where the input $u(v)$ has been turned into a local input $u'(v)$ at the same location as x , and where the forwarder allows one output on u to interact with u' instead. The key observation is that the linear forwarder $u \multimap u'$ is easy to implement: it is just a packet containing two Uniform Resource Locators (URLs in the following) and directed to the location of u . Using this mechanism, the PiDuce machine admits to import a remote (PiDuce) service and to perform

inputs on it.

The second motivation of the PiDuce project is to design a distributed machine running applications that may be exported to the Web (Web services). For this reason we use the standard formats XML [32] and WSDL [29,30] for describing values and interfaces, respectively. PiDuce programs may construct XML documents and, by means of a pattern matching mechanism, deconstruct them. The compiler performs a semantic analysis guaranteeing that invalid documents can never be produced.

The design of the PiDuce datatype and pattern languages, as well as most of the algorithms regarding these features, have been strongly influenced by the XDuce [21] and CDuce [5] prototypes – two functional languages with native XML datatypes. With respect to these languages, a major technical difficulty in the PiDuce datatype language is that values, as in XML, may contain channels that are URLs where values can be sent. A channel is represented by the WSDL interface describing the schema of the values it accepts. The semantics rules expose an environment that is partially supplied by local service declarations and partially by the global environment. The maintenance of this environment means that communications also gather information about the schemas of the channels contained in the message. A related problem is found in the algorithm matching a document against a pattern (pattern matching). The algorithm checks if the document conforms with the schema specified in the pattern and returns a set of variable bindings. As in XDuce, pattern matching in PiDuce is implemented using top-down tree automata, but the presence of channels inside values increases the complexity of the algorithm. In particular, verifying that a channel matches a pattern reduces to checking whether the schema of the channel is a subschema of the one specified in the pattern or not. This, in general, requires exponential time in the size of the tree automata of the pattern [12] and may significantly degrade the run-time efficiency of possible implementations. To alleviate this problem we add some restrictions on schemas that make subschema verification polynomial [10].

The result of our motivations – the PiDuce prototype – is a formally specified distributed machine running programs that are (statically typed π -calculus) processes with XML values, datatypes, and patterns. The prototype includes two interacting modules: the runtime environment and the Web interface (see Figure 1). The former realizes a type-safe environment: every operation performed therein can never manifest a type error. The latter decouples a large part of the prototype from the actual transport protocols and technologies used in distributed communication. In this way the runtime environment may be adapted to different contexts without effort. In particular, the Web interface defines the interface between PiDuce services and Web services. This interface includes functions for encoding PiDuce schemas into XML-Schema [26–28] and *vice versa*. Although XML-Schema and PiDuce schemas have a significant com-

mon intersection, they cannot be compared in terms of expressivity. There are features of XML-Schema (such as keys, references, facets, derivation by extension, derivation by restriction, substitution groups, the keywords final and block, the interleaving operator) that are not supported by PiDuce schemas and, conversely, features of PiDuce schemas that cannot be mapped in XML-Schemas (such as channels, a subclass of mutual recursive definitions).

As it stands, the PiDuce prototype is a programming technology for defining and experimenting Web services. Compared to an emerging standard, the BPEL language [14], the PiDuce language has mostly the same operations, except for exception handling and transactions. These operations are definitely relevant for Web services, but their formal account and runtime support is outside the scope of this paper. A discussion of this issue is undertaken in the conclusions.

This contribution is structured as follows. Section 2 is an introduction to the language constructs through examples. Section 3 defines the syntax of the PiDuce programming language. Section 4 defines the subschema relation and the static semantics of the PiDuce language. Section 5 describes the operational semantics of local operations. Section 6 defines the PiDuce distributed machine and the static and dynamic semantics of instructions that deal with remote locations. Section 7 describes the architecture of the PiDuce machine, and in particular the channel manager and the virtual machine. Section 8 describes the Web interface and how interoperability is achieved in the PiDuce prototype. Section 9 describes the compiler and it illustrates the PiDuce object code. Section 10 describes some extensions of PiDuce. Section 11 discusses related works and we conclude in Section 12.

2 Getting started

The basic elements of PiDuce are introduced by means of a sequel of examples. The formal presentation is deferred to the next section.

Values in PiDuce represent XML documents. For example, the document

```
<msg>hello</msg><doc/>
```

is written `msg["hello"],doc[]`.

Processes in PiDuce are intended to define Web services. For example, a printer service that collects color and black-white printing requests is defined as follow

```

print?*( x : Pdf + JPeg )
  match x with
    y : Pdf --> printbw!( y )
    z : JPeg --> printc!( z )

```

The `print` service checks whether the received value `x` matches `Pdf` or `JPeg`; in the first case it forwards the value `x` to the black-white printer, in the second case it forwards the value `x` to the color printer. The basic mechanism for interactions is message passing. For example `print!(document)` invokes the service `print` passing it the value `document`. Service invocation is non-blocking and asynchronous: the sender does not wait that the receiver really consumes the message. The star after the question mark of `print` indicates that this service is replicated: the process is capable of handling an unlimited number of requests.

The parallel execution of several activities is defined by the `spawn` construct. For example

```

spawn { print!( document1 ) } print!( document2 )

```

invokes the service `print` twice. Because of asynchrony, there is no guarantee as to which invocation will be served first. More elaborated forms of communication, such as *rendez-vous* and sequentiality, can be programmed using explicit continuation-passing style.

PiDuce also provides a `select` operation that is similar to the homonymous system call in socket programming, to the “pick activity” in BPEL, and to the *input-guarded choice* in π -calculus. In its simplest form the operation allows one to define ephemeral services that serve exactly one request and disappear. For example, the above `print` channel may be made ephemeral as follows

```

eph_print?( x : Pdf + JPeg )
  match x with
    y : Pdf --> printbw!( y )
    z : JPeg --> printc!( z )

```

(note the missing `*` after `eph_print?`). In general, the `select` operation groups several input operations to be executed in mutual exclusion. For instance, the following `select` waits either a print request or a fax request; once one of these requests arrives, the corresponding continuation is performed and the service terminates:

```

select {
  print?( x : Pdf + JPeg ) ... // PRINT
  fax?( x : Document ) ... // FAX
}

```

Service names, called *channel literals* in PiDuce, can be created dynamically. For example

```
new print : (Pdf + JPeg)0 in P
```

creates a fresh `print` channel literal that handles either `Pdf` or `JPeg` values. The scope of the declaration is restricted to P . Creating a new channel amounts to making a service available at a URL address that is the one of the runtime environment executing the `new` instruction (plus a unique local path). The channel `print` is published as a WSDL, which declares that the service may be invoked with documents of schema `Pdf + JPeg` (“+” is the union schema in XML-Schema) and the interaction pattern is one-way (the *capability* “0”). Actually, channels are first-class citizens in PiDuce: they are values that can be sent over and received from other channels and they can be examined by pattern matching. With this standpoint, the operation `new` is intended to declare the schema of a channel literal.

Schemas in PiDuce approximate XML-Schemas. For example, the XML-Schema

```
<xsd:element name="a" type="xsd:integer"/>
```

is written as `a[int]` and describes values consisting of an element labelled `a` and whose content is an integer. More complex XML-Schemas – as sequences and choices – can be also written in PiDuce. For instance

```
<xsd:sequence>
  <xsd:element name="a" type="xsd:integer"/>
  <xsd:choice>
    <xsd:element name="b" type="xsd:string"/>
    <xsd:element name="c"/>
  </xsd:choice>
</xsd:sequence>
```

is written as `a[int], (b[string] + c[])`. XML-Schemas may also describe documents with repeated structure, such as

```
<xsd:sequence minOccurs="0" maxOccurs="unbound">
  <xsd:element name="a"/>
</xsd:sequence>
```

representing a possibly empty sequence of empty, `a`-labelled elements. Schemas with a repeated structure are written in PiDuce by means of recursion. For example, the previous schema is defined by:

$$U = a[], U + ()$$

where `()` is the empty document. PiDuce schemas have also features that are at

odd with XML-Schema, and conversely. A detailed discussion of the relationship between XML and PiDuce schemas is undertaken in Section 8.

In the previous codes for `print` and `eph_print`, the pattern matching construct has been used to cast the received value to schemas `File` or `Picture`. Pattern matching is problematic in PiDuce because values may be channel literals. In such cases, matching a channel with a schema S amounts to downloading the WSDL of the channel and verifying its “conformance” with S . Conformance is formally defined by the *subschema*, which usually verifies whether the values described by a schema are a subset of those described by another one. In the case of channels, the subschema reduces to a subschema on the schema of the values carried by the channels taking into account their *capability*, that is how channels are used by the receivers. The subschema is fundamental for checking the type safety of PiDuce processes performed at compile time. For example, the process `print!(document)` in the scope of the above definition of `print` is correct as long as the schema of `document` is either `Pdf` or `Jpeg`. The details about the subschema relation and the type checking are discussed in Section 4.

3 The PiDuce programming language

The syntax of PiDuce includes the categories *labels*, *expressions*, *schemas*, *patterns*, and *processes* that are defined in Table 1. Several countably infinite sets are used: the set of *tags*, ranged over by a, b, \dots ; the set of *channel literals* (i.e. URLs) ranged over by u, v, \dots ; the set of *variables*, ranged over by x, y, z, \dots ; the set of *schema names*, ranged over by U, V, \dots . We use u, v, \dots to range over channel literals and variables.

In the following, variables or channel literals u in $u!(E)$, $u?(F)$ and $u?*(F)$ are called *subjects*.

The sets $\mathbf{fn}(\cdot)$ of *free variables* and $\mathbf{bn}(\cdot)$ of *bound variables* are defined for expressions, patterns, and processes as follows:

$\mathbf{fn}(E)$: is the set of variables occurring in E ;
 $\mathbf{fn}(F)$: is the set of variables occurring in F ;
 $\mathbf{fn}(P)$: is the set of variables occurring in P that are not *bound*.

An occurrence of x in P is *bound* in a branch $u?(F)$ P of a select or in the replicated input $u?*(F)$ P if $x \in \mathbf{fn}(F)$; an occurrence of u in P is *bound* in `new u : $\langle S \rangle^\kappa$ in P` . The definitions of alpha-conversion and substitution for bound variables are standard. In the whole paper, we identify terms that are equal up-to alpha-conversion.

Table 1

PiDuce syntax (\mathbf{T} includes `int`, `string`, integer and string constants).

$L ::=$	label	$S ::=$	schema
a	(tag)	\perp	(empty schema)
\sim	(wildcard label)	$()$	(void schema)
$L + L$	(union)	\mathbf{T}	(base type)
$L \setminus L$	(difference)	$\langle S \rangle^\kappa$	(channel schema)
$E ::=$	expression	$L[S], S$	(sequence schema)
$()$	(void)	$S + S$	(union schema)
\mathbf{n}	(integer constant)	\mathbf{U}	(schema name)
\mathbf{s}	(string constant)	$P ::=$	process
\mathbf{u}	(channel literal)	$\mathbf{0}$	(nil)
x	(variable)	$u!(E)$	(output)
$E \text{ op } E$	(operations)	$\text{select } \{u_i?(F_i) P_i \mid i \in 1..n\}$	(select)
$a[E], E$	(sequence)	$\text{new } u : \langle S \rangle^\kappa \text{ in } P$	(new)
$F ::=$	pattern	$\text{match } E \text{ with } \{F_i \rightarrow P_i \mid i \in 1..n\}$	(match)
S	(schema)	$\text{spawn } \{P\} P$	(spawn)
$u : F$	(variable pattern)	$u?*(F) P$	(replication)
$L[F], F$	(sequence pattern)		

Labels. Labels specify collections of tags. Let \mathcal{L} be the set of all tags. The semantics of labels is defined by the following function $\widehat{\cdot}$:

$$\widehat{a} = \{a\} \quad \widehat{\sim} = \mathcal{L} \quad \widehat{L + L'} = \widehat{L} \cup \widehat{L'} \quad \widehat{L \setminus L'} = \widehat{L} \setminus \widehat{L'}$$

We write $a \in L$ for $a \in \widehat{L}$.

Expressions. Expressions are $()$, integer constants, string constants, channel literals, variables, basic operations, or sequences of labelled values. For simplicity we do not permit the creation of a sequence E, E' even if E is a tagged expression. It is always necessary to explicitly provide a label as in $a[E], E'$. The details about basic operations `op` are omitted (they include functions such as $+$, $-$, $=$, $>$, etc.); we assume that every `op` has an associated type $\mathbf{T} \times \mathbf{T}' \rightarrow \mathbf{T}''$, where \mathbf{T} , \mathbf{T}' , and \mathbf{T}'' are base types. For example

$$a[\mathbf{i} - \mathbf{j}], b[\text{"the"} + \mathbf{x}]$$

is an expression evaluating to a sequence of two labelled elements where the first contains the difference of the values associated to the integers variables `i` and `j`, and the second contains the concatenation of `"the"` and the string stored in `x`.

Nonempty sequences always consist of tagged elements, except possibly the last one. For example $5, a["the"], ()$ and $a["the"], (), 5$ are wrong, while $a[5], b["the"], ()$ and $a["the"], b[()], x$ are correct. In the following, whenever possible $()$ is omitted: values such as $a[()]$ and $a[E], ()$ are shortened into $a[]$ and $a[E]$, respectively.

Channel literals are service references, that is URL addresses of WSDL interfaces, such as `http://www.cs.unibo.it/PiDuce.wsdl`. The WSDL file contains: (i) the location where the channel is available, (ii) the kind of messages it accepts, (iii) the protocol that must be used for sending messages, (iv) and the operations it supports (input, output, input/output) – see Section 8. This information is used by the runtime environment for validation, pattern matching, and communication.

Among expressions we distinguish *values*, which are expressions that do not contain operators. Let V range over values. Expressions are evaluated into values by means of the following rules (op is the semantic operator for op):

$$\frac{n' \text{ op } n'' = n}{n' \text{ op } n'' \Downarrow n} \qquad \frac{s' \text{ op } s'' = s}{s' \text{ op } s'' \Downarrow s} \qquad \frac{E \Downarrow V \quad E' \Downarrow V'}{a[E], E' \Downarrow a[V], V'}$$

Values may contain variables. However, variables occurring in values that are exchanged at runtime represent channel literals that have not been published (because processes are closed with respect to variables of other schemas, *cf.* “channeled environments” in Section 6).

Schemas. Schemas describe collections of values that are structurally similar. The syntax of schemas includes the category of base types T that, in this paper, are integers `int`, strings `string` and integer and string constants n and s , respectively. The base types n and s represent the sets $\{n\}$ and $\{s\}$, respectively. The schema \perp describes the empty set of documents; $()$ describes the empty document. The schema $\langle S \rangle^\kappa$ describes channels that carry messages of schema S and that may be used with *capability* $\kappa \in \{I, O, IO\}$. The capabilities I, O, IO mean that the channel can be used for performing inputs, outputs, and both inputs and outputs, respectively. For example $\langle \text{int} \rangle^0$ describes the set of channel literals that may be invoked with integers. The schema $L[S], S'$ describes sequences where the first document starts with a tag in L containing a document of schema S and the remaining part is of schema S' . For example,

$$a[\text{int}], (\sim \setminus a)[\text{string}], ()$$

describes all the documents made of an a -labelled element containing an integer, followed by a “non- a ”-labelled element containing a string. Nonempty sequence schemas consist of labelled elements, except possibly the last one. In what follows $L[()]$ and $L[S], ()$ are shortened into $L[]$ and $L[S]$, respectively.

The schema $S + S'$ describes the set of documents belonging to either S or to S' . Schemas include schema names that are bound by global definitions (see the following paragraph about programs): $U_1 = S_1 \ ; \ ; \ \dots \ ; \ ; \ U_n = S_n$. Recursion and mutual recursion are admitted. For example,

$U = a[V], U + () \ ; \ ; \ V = \text{int}$

defines sequences of a -labelled integers. The following definitions will be used in the rest of the paper (and are part of the **PiDuce** environment):

$\text{Empty} = \text{Empty} \ ; \ ;$
 $\text{AnyChan} = \langle \text{Empty} \rangle^0 + \langle \text{Any} \rangle^1 \ ; \ ;$
 $\text{Any} = () + \text{int} + \text{string} + \text{AnyChan} + \sim[\text{Any}], \text{Any}$

The schema **Empty** describes an empty set of values and is equivalent to \perp (\perp is included in the syntax to ease the presentation of the subschema relation in Section 4). **AnyChan** describes every channel value. In particular $\langle \text{Empty} \rangle^0$ collects any output channel because of the contravariance of the output channel constructor whilst $\langle \text{Any} \rangle^1$ collects any input channel because of the covariance of the input channel constructor (channels having input/output capability are matched by both branches). Finally, **Any** collects all the values.

PiDuce schemas correspond to regular tree grammars [12]. This class of languages retains a subschema relation that is decidable (this operation is fundamental because it is used in type checking and during pattern-matching).

Patterns. Patterns permit the deconstruction of values using matching. The pattern S is matched by values of schema S . A variable $x : F$ can be bound in the course of matching to a value that in turn matches the sub-pattern F . For example, $u : a[] + b[]$ may bind values such as $a[]$ or $b[]$; $u : \langle a[] \rangle^0$ may bind any channel value of schema $\langle a[] \rangle^0$ but it does not bind a channel value of schema $\langle a[] + b[] \rangle^0$ – see the definition of pattern matching in Section 5 –; the pattern $u : a[v : b[]]$ binds two variables at the same time: u to values of schema $a[b[]]$ and v to values of schema $b[]$. The pattern $L[F], F'$ is matched by values of the form $a[V], V'$, with $a \in L$ and F and F' matched by V and V' , respectively. The pattern $L[F], F'$ is also matched by values $a[V]$ if $a[V]$ matches with $L[F]$ and $()$ matches with F' . As usual, $L[F], ()$ is shortened into $L[F]$.

Patterns in **PiDuce** are *linear with respect to variables*: variables occurring in $u : F$ and $L[F], F'$ do not clash. Because of linearity, if the pattern matching algorithm succeeds it yields exactly one binding for each variable.

Patterns retain a schema that may be obtained by removing every pattern

variable. This deletion operation is formalized by the function $\mathbf{schof}(\cdot)$ below:

$$\begin{aligned}\mathbf{schof}(S) &= S \\ \mathbf{schof}(u : F) &= \mathbf{schof}(F) \\ \mathbf{schof}(L[F], F') &= L[\mathbf{schof}(F)], \mathbf{schof}(F')\end{aligned}$$

Processes. Processes are the computing entities of PiDuce. $\mathbf{0}$ is the idle process; $u!(E)$ evaluates E to a value and outputs it on the channel u . This means that $u!(E)$ is idle if E cannot be evaluated. The process $\mathbf{select} \{u_i?(F_i) P_i \mid i \in 1..n\}$ inputs on the channel u_i a value that matches with F_i yielding a substitution σ and behaves as $P_i\sigma$. We always abbreviate $\mathbf{select} \{u?(F) P\}$ to $u?(F) P$. The process $\mathbf{new} u : \langle S \rangle^\kappa \mathbf{in} P$ permits the definition of a fresh channel u and binds it within the continuation P . In the continuation P , u may be used as subject of outputs and inputs. The capability κ is the one that is exposed when the channel is extruded – it is the capability that is declared in the WSDL of the channel. The process $\mathbf{match} E \mathbf{with} \{F_i \rightarrow P_i \mid i \in 1..n\}$ tests whether the value to which E evaluates matches one of the patterns F_i 's. The order of the branches is relevant, so that the first matching pattern determines the continuation (first match semantics). If the match of F_k succeeds, the continuation $P_k\sigma$ is run, where σ is the substitution yielded by the pattern matching algorithm. The process $\mathbf{spawn} \{P\} Q$ spawns the execution of P on a separate thread and continues as Q . The replicated input $u?*(F) P$ consumes a message on u , it spawns the continuation $P\sigma$, where σ is the substitution yielded by matching the message with the pattern F , and then it becomes available for other messages on u .

Inputs in \mathbf{select} and replicated inputs are always *exhaustive*. More precisely, F in $u?*(F) P$ and in every branch $u?(F) P$ of a \mathbf{select} must match every possible value carried by u . It is worth noticing that invalid messages (those that do not conform with the schema of the channel) are automatically discarded by the Web interface – see Section 8. This circumstance never occurs between interacting threads in the same PiDuce process because of the static typechecking.

The syntax of Table 1 does not include PiDuce operations that affect remote machines. In PiDuce it is possible to create channel literals at remote locations, to input on remote channel literal and to import remote channels or Web services. We discuss these operations and their semantics in Section 6.

Programs. A PiDuce program is $U_1 = S_1;; \dots;; U_n = S_n;; P$, namely a sequence of schema name definitions and a process. For simplicity we assume that schema names U_1, \dots, U_n are pairwise different. A sequence of schema name definitions may be represented by a map with finite domain \mathcal{E} that

takes a schema name and returns the associated schema. In the following, we always use this representation.

4 The subschema relation and the type system

The subschema relation of **PiDuce** is similar to the one developed for **XDuce** [22], except for the rules accounting for channel schemas. When schemas do not have channels, the subschema relation calculates tree language inclusion (a decidable operation, although computationally expensive, because schemas in **XDuce** and **PiDuce** are regular tree languages [12]).

Let \leq be the smallest reflexive relation on capabilities containing $\mathbf{IO} \leq \mathbf{0}$ and $\mathbf{IO} \leq \mathbf{I}$. Let $\wp(I)$, where I is a set, be the powerset of I .

Definition 1 *The subschema relation $\lesssim_{\mathbf{A}}$ is the smallest relation closed under reflexivity, commutativity of unions, with \perp as identity of unions, and under the rules in Table 2. We abbreviate $S \lesssim_{\mathbf{A}} T \Rightarrow \mathbf{A}'$ into $S \lesssim_{\mathbf{A}} T$.*

Let \lesssim be the least preorder containing $S \lesssim_{\emptyset} T$. If $S \lesssim T$ we say that S is a subschema of T ; if $S \lesssim T$ and $T \lesssim S$ we say that S and T are (subschema) equivalent.

The subschema relationship between S and T is demonstrated by a proof tree with a conclusion $S \lesssim_{\mathbf{A}} T \Rightarrow \mathbf{A}'$, where $\mathbf{A} = \emptyset$. In this proof tree the \mathbf{A} is filled with pairs (\mathbf{U}, S) , where the first element is always a schema name and the second one is a schema. The presence of (\mathbf{U}, S) in \mathbf{A} means that the subschema relation between \mathbf{U} and S has been proved or is being proved. The set \mathbf{A}' contains all the pairs (\mathbf{U}, S) such that the subschema relation between \mathbf{U} and S has been proved. We discuss the rules in order.

Rules (BOT) states that \perp is the smallest schema; rules (LBOT), (HBOT), and (TBOT) establish that $L[S], S'$ is a subschema of \perp – therefore equivalent to it – if either \hat{L} is empty, or one between S and S' is equivalent to \perp . Rules (INT) and (STRING) define that integer and string constants are subschema of **int** and **string**, respectively.

Rules (CHAN-I), (CHAN-O), and (CHAN-IO) reduce subschema to the arguments of the channel constructors; they respectively establish covariant, contravariant, and invariant relationships on the arguments. Intuitively these relationships can be explained as follows: if $u : \langle S \rangle^{\mathbf{I}}$ and $\langle S \rangle^{\mathbf{I}} \lesssim \langle T \rangle^{\mathbf{I}}$ then we are allowed to use u where a channel of type $\langle T \rangle^{\mathbf{I}}$ is expected. In order to guarantee type safety of any process performing input operations on u believing that it has type $\langle T \rangle^{\mathbf{I}}$, it must be that $S \lesssim T$ (covariance). If $v : \langle S \rangle^{\mathbf{0}}$ and

Table 2

Subschema relation in PiDuce.

$\stackrel{(\text{BOT})}{\perp \lesssim_{\mathbf{A}} T \Rightarrow \mathbf{A}}$	$\stackrel{(\text{LBOT})}{\frac{\widehat{L} = \emptyset}{L[S], S' \lesssim_{\mathbf{A}} \perp \Rightarrow \mathbf{A}}}$	$\stackrel{(\text{HBOT})}{\frac{S \lesssim_{\mathbf{A}} \perp \Rightarrow \mathbf{A}'}{L[S], S' \lesssim_{\mathbf{A}} \perp \Rightarrow \mathbf{A}'}}$	$\stackrel{(\text{TBOT})}{\frac{S' \lesssim_{\mathbf{A}} \perp \Rightarrow \mathbf{A}'}{L[S], S' \lesssim_{\mathbf{A}} \perp \Rightarrow \mathbf{A}'}}$
	$\stackrel{(\text{INT})}{\mathbf{n} \lesssim_{\mathbf{A}} \mathbf{int} \Rightarrow \mathbf{A}}$	$\stackrel{(\text{STRING})}{\mathbf{s} \lesssim_{\mathbf{A}} \mathbf{string} \Rightarrow \mathbf{A}}$	
$\stackrel{(\text{CHAN-I})}{\frac{\kappa \leq \mathbf{I} \quad S \lesssim_{\mathbf{A}} T \Rightarrow \mathbf{A}'}{\langle S \rangle^{\kappa} \lesssim_{\mathbf{A}} \langle T \rangle^{\mathbf{I}} \Rightarrow \mathbf{A}'}}$	$\stackrel{(\text{CHAN-O})}{\frac{\kappa \leq \mathbf{O} \quad T \lesssim_{\mathbf{A}} S \Rightarrow \mathbf{A}'}{\langle S \rangle^{\kappa} \lesssim_{\mathbf{A}} \langle T \rangle^{\mathbf{O}} \Rightarrow \mathbf{A}'}}$	$\stackrel{(\text{CHAN-IO})}{\frac{S \lesssim_{\mathbf{A}} T \Rightarrow \mathbf{A}' \quad T \lesssim_{\mathbf{A}'} S \Rightarrow \mathbf{A}''}{\langle S \rangle^{\mathbf{IO}} \lesssim_{\mathbf{A}} \langle T \rangle^{\mathbf{IO}} \Rightarrow \mathbf{A}''}}$	
	$\stackrel{(\text{UNIONR})}{\frac{S \lesssim_{\mathbf{A}} T \Rightarrow \mathbf{A}'}{S \lesssim_{\mathbf{A}} T + T' \Rightarrow \mathbf{A}'}}$	$\stackrel{(\text{UNIONL})}{\frac{S \lesssim_{\mathbf{A}} T \Rightarrow \mathbf{A}' \quad S' \lesssim_{\mathbf{A}'} T \Rightarrow \mathbf{A}''}{S + S' \lesssim_{\mathbf{A}} T \Rightarrow \mathbf{A}''}}$	
$\stackrel{(\text{SPLIT})}{\frac{\widehat{L} \not\subseteq \widehat{L}' \quad (L \setminus L')[S], S' \lesssim_{\mathbf{A}} T'' \Rightarrow \mathbf{A}' \quad (L \cap L')[S], S' \lesssim_{\mathbf{A}'} L'[T], T' + T'' \Rightarrow \mathbf{A}''}{L[S], S' \lesssim_{\mathbf{A}} L'[T], T' + T'' \Rightarrow \mathbf{A}''}}$			
$\stackrel{(\text{LSEQ})}{\frac{(\widehat{L} \subseteq \widehat{L}_i)^{i \in I} \quad \wp(I) = \{J_1, \dots, J_n\} \quad \left(S \lesssim_{\mathbf{A}_{k-1}} \sum_{i \in J_k} T_i \Rightarrow \mathbf{A}_k \quad \text{or} \quad S' \lesssim_{\mathbf{A}_{k-1}} \sum_{i \in I \setminus J_k} T'_i \Rightarrow \mathbf{A}_k \right)^{k \in 1..n}}{L[S], S' \lesssim_{\mathbf{A}_0} \sum_{i \in I} L_i[T_i], T'_i \Rightarrow \mathbf{A}_n}}$			
$\stackrel{(\text{NAMEL})}{\frac{(\mathbf{U}, T) \in \mathbf{A}}{\mathbf{U} \lesssim_{\mathbf{A}} T \Rightarrow \mathbf{A}}}$	$\stackrel{(\text{NAMEH})}{\frac{\mathbf{A}' = \mathbf{A} \cup \{(\mathbf{U}, T)\} \quad \mathcal{E}(\mathbf{U}) \lesssim_{\mathbf{A}'} T \Rightarrow \mathbf{A}''}{\mathbf{U} \lesssim_{\mathbf{A}} T \Rightarrow \mathbf{A}''}}$	$\stackrel{(\text{NAMER})}{\frac{S \lesssim_{\mathbf{A}} \mathcal{E}(\mathbf{U}) + T \Rightarrow \mathbf{A}'}{S \lesssim_{\mathbf{A}} \mathbf{U} + T \Rightarrow \mathbf{A}'}}$	

$\langle S \rangle^0 \lesssim \langle T \rangle^0$ then we are allowed to use v where a channel of type $\langle T \rangle^0$ is expected. A process performing output operations on v believing that it has type $\langle T \rangle^0$ does not do anything illegal as long as $T \lesssim S$ (contravariance). The combination of these two rules results into the invariance of channel schemas with IO capability.

Rule (UNIONR) allows us to drop branches of unions on the right; (UNIONL) allows us to reduce the subschema relation for a union schema on the left to the subschema of every of its branches. Rule (UNIONR) is not sufficient to adequately define the subschema relation when the schema on the right is a union. For example, consider the schemas $(a + b)[S], T$ and $a[S], T + b[S], T$. Then $(a + b)[S], T$ is neither a subschema of $a[S], T$ nor of $b[S], T$, but it is a subschema of their union. Rule (SPLIT) deals exactly with such relations.

Still, rules (UNIONR) and (SPLIT) leave a number of cases out. For example the schema $a[S + S'], T$ is a subschema of $a[S], T + a[S'], T$ but this relation cannot be inferred without using (LSEQ). To clarify (LSEQ), let us admit a schema intersection operator \cap such that $S \cap T$ is the schema describing the values that belong to both S and T . Then $L[S], T$ may be rewritten as $L[S], \text{Any} \cap \sim[\text{Any}], T$. Therefore

$$\begin{aligned} L_1[S_1], T_1 + L_2[S_2], T_2 &= (L_1[S_1], \text{Any} \cap \sim[\text{Any}], T_1) + (L_2[S_2], \text{Any} \cap \sim[\text{Any}], T_2) \\ &= (L_1[S_1], \text{Any} + L_2[S_2], \text{Any}) \cap (\sim[\text{Any}], T_1 + \sim[\text{Any}], T_2) \\ &\quad \cap (L_1[S_1], \text{Any} + \sim[\text{Any}], T_2) \cap (\sim[\text{Any}], T_1 + L_2[S_2], \text{Any}) \end{aligned}$$

where the last equality follows by distributivity of \cap with respect to union. Next, if one intends to derive $L[S], T \lesssim L_1[S_1], T_1 + L_2[S_2], T_2$ when $\widehat{L} \subseteq \widehat{L}_1 \cap \widehat{L}_2$, by the above arguments it is possible to rewrite the check as follows, remembering that **Any** is the greatest schema:

$$\text{for every } J \subseteq \{1, 2\} \text{ either } S \lesssim \sum_{j \in J} S_j \text{ or } T \lesssim \sum_{j \in \{1, 2\} \setminus J} T_j$$

This is exactly the bottom premise of (LSEQ) when $I = \{1, 2\}$. In (LSEQ) the subsets of I have been ordered for accumulating the pairs (U, S) in the sets **A**. A particular case of (LSEQ) is when $I = \{1\}$. For example when one intends to prove that $a[S], T \lesssim (a + b)[S'], T'$. In this case the subsets of I are \emptyset and $\{1\}$ and we are reduced to prove:

$$\left(S \lesssim \perp \text{ or } T \lesssim T' \right) \text{ and } \left(S \lesssim S' \text{ or } T \lesssim \perp \right)$$

That is, when $S \not\lesssim \perp$ and $T \not\lesssim \perp$, we are reduced to $S \lesssim S'$ and $T \lesssim T'$.

The last three rules are about schema names. Rule (NAMEL) derives a relation $U \lesssim_A T$ if the pair (U, T) is in the (hypothesis) set **A**. Rule (NAMEH) is the unique one that uses an augmented set in the hypotheses. In order to prove $U \lesssim_A T$, one unfolds U and, at the same time, records in the set **A** that $U \lesssim_A T$ is being proved. This way loops are avoided: if, during the proof of $U \lesssim_A T$, one reduces to $U \lesssim_{A'} T$ then it is possible to terminate (by rule (NAMEL)). This is the case, for example, when $U \lesssim_{\emptyset} V$ must be proved, with $\mathcal{E}(U) = V$ and $\mathcal{E}(V) = V$. Rule (NAMER) unfolds schema names that occur on the right. It is worth to notice that, in order to verify $\text{int} \lesssim_A U$, where $\mathcal{E}(U) = \text{int}$, it suffices to replace U with the equivalent schema $U + \perp$ and then apply (NAMER).

Remark 2 *We conjecture that the relation \lesssim_{\emptyset} is closed by transitivity. (We have not been able to provide a direct proof.) In this case, the relation \lesssim_{\emptyset} should be equal to \lesssim . In [10] the transitivity of \lesssim_{\emptyset} for labelled-determined schemas follows by its coincidence with a simulation relation that is transitive by definition.*

The subschema plays a prominent role in the static semantics of PiDuce – the type system. Few preliminary notations are introduced. Let Γ, Δ , called *environments* be maps that take variables and channel literals and return schemas. We write $\text{dom}(\Gamma)$ for the set of names in the domain of Γ . Let $\Gamma + \Delta$ be $(\Gamma \setminus \text{dom}(\Delta)) \cup \Delta$, where $\Gamma \setminus X$ removes from Γ all the bindings of names in X . Let $\Gamma|_X$ the environment that is equal to Γ for variables and channel literals in $\text{dom}(\Gamma) \cap X$ and it is undefined otherwise. Finally, let $(\Gamma; \Delta) + F$ be the pair $\Gamma + \text{Env}(F); \Delta \setminus \text{dom}(\text{Env}(F))$, where $\text{Env}(\cdot)$ is the following function:

$$\begin{aligned} \text{Env}(S) &= \emptyset \\ \text{Env}(u : F) &= u : \text{schof}(F) + \text{Env}(F) \\ \text{Env}(L[F], F') &= \text{Env}(F) + \text{Env}(F') \end{aligned}$$

The judgments $\Gamma \vdash E : S$ – read E has schema S in the environment Γ – and $\Gamma; \Delta \vdash P$ – read P is well typed in the environment Γ and local environment Δ – are the least relations satisfying the rules in Table 3.

Rules for expressions, (NIL) and (SPAWN) are standard. Rule (OUT) types outputs. By definition of subschema, the premise $T \lesssim \langle S \rangle^0$ entails that u has a channel schema. In particular, if u has been locally defined – $u \in \text{dom}(\Delta)$, see rule (NEW) – the judgment $\Gamma + \Delta \vdash u : T$ entails that $T = \langle T' \rangle^{10}$, for some T' . Otherwise $u \in \text{dom}(\Gamma) \setminus \text{dom}(\Delta)$, that means u is a channel bound in a pattern matching operation. In this case, again by definition of subschema, u must have been declared with a capability $\kappa \leq 0$. Note that u can be typed as a union of channel schemas, for example $u : \langle \mathbf{a}[] \rangle^0 + \langle \mathbf{b}[] \rangle^0$. When this is the case, V must be a subschema of the smallest schema carried by u . In the example, it is not possible to send values over u because the type system requires a value which is a subschema of both $\mathbf{a}[]$ and $\mathbf{b}[]$. If we consider $u : \langle \mathbf{a}[] \rangle^0 + \langle \mathbf{a}[] + \mathbf{b}[] \rangle^0$ we can send the value $\mathbf{a}[]$ over u because $\mathbf{a}[]$ is a subschema of both $\mathbf{a}[]$ and $\mathbf{a}[] + \mathbf{b}[]$.

Rule (SELECT) types input-guarded choices. The first hypothesis types subjects. The second hypothesis types the continuation of every summand in the environment $\Gamma; \Delta$ plus that defined by the pattern. The third hypothesis checks the exhaustiveness of every pattern. As for outputs the hypothesis $S_i \lesssim \langle \text{schof}(F_i) \rangle^1$ does not strictly require u_i to be a channel schema. In particular u_i can be a union of channel schemas and, in this case, the schema of the pattern must be a superschema of every schema that can be received over u_i . For instance if $u : \langle \mathbf{a}[] \rangle^1 + \langle \mathbf{b}[] \rangle^1$ then the input $u?(\mathbf{a}[] + \mathbf{b}[]) P$ is well typed because $\mathbf{a}[] + \mathbf{b}[]$ is a superschema of both $\mathbf{a}[]$ and $\mathbf{b}[]$. Rule (REPIN) is similar to (SELECT) but the subject is checked to be local.

Rule (NEW) types **new** $u : \langle S \rangle^\kappa$ in P in $\Gamma; \Delta$ provided P is typable with in $\Gamma + u : \langle S \rangle^\kappa; \Delta + u : \langle S \rangle^{10}$. The first component of the pair of environments is extended with the exported schema $\langle S \rangle^\kappa$ of the channel; this definition is used

Table 3
Typing rules for PiDuce.

Expressions :

$$\begin{array}{c} \Gamma \vdash () : () \quad \Gamma \vdash n : n \quad \Gamma \vdash s : s \quad \frac{\Gamma(u) = S}{\Gamma \vdash u : S} \\[2ex] \frac{\Gamma \vdash E : S \quad \Gamma \vdash E' : S'}{\Gamma \vdash a[E], E' : a[S], S'} \quad \frac{\Gamma \vdash E_1 : T_1 \quad \Gamma \vdash E_2 : T_2 \quad T_1 \lesssim T'_1 \quad T_2 \lesssim T'_2 \quad \text{op} : T'_1 \times T'_2 \rightarrow T}{\Gamma \vdash E_1 \text{ op } E_2 : T} \end{array}$$

Processes :

$$\begin{array}{c} \text{(NIL)} \quad \Gamma; \Delta \vdash 0 \quad \text{(OUT)} \quad \frac{\Gamma \vdash E : S \quad \Gamma + \Delta \vdash u : T \quad T \lesssim \langle S \rangle^0}{\Gamma; \Delta \vdash u!(E)} \\[2ex] \text{(SELECT)} \quad \frac{(\Gamma + \Delta \vdash u_i : S_i \quad (\Gamma; \Delta) + \text{Env}(F_i) \vdash P_i \quad S_i \lesssim \langle \text{schof}(F_i) \rangle^I)^{i \in 1..n}}{\Gamma; \Delta \vdash \text{select } \{u_i?(F_i)P_i\}^{i \in 1..n}} \\[2ex] \text{(NEW)} \quad \frac{\Gamma + u : \langle S \rangle^\kappa; \Delta + u : \langle S \rangle^{I0} \vdash P}{\Gamma; \Delta \vdash \text{new } u : \langle S \rangle^\kappa \text{ in } P} \\[2ex] \text{(MATCH)} \quad \frac{\Gamma + \Delta \vdash E : S \quad ((\Gamma; \Delta) + \text{Env}(F_i) \vdash P_i)^{i \in 1..n} \quad S \lesssim \sum_{i \in 1..n} \text{schof}(F_i) \quad (\text{schof}(F_i) \lesssim S)^{i \in 1..n-1} \quad (\text{schof}(F_j) \not\lesssim \sum_{j < i} \text{schof}(F_j))^{j \in 2..n}}{\Gamma; \Delta \vdash \text{match } E \text{ with } \{F_i \rightarrow P_i\}^{i \in 1..n}} \\[2ex] \text{(SPAWN)} \quad \frac{\Gamma; \Delta \vdash P \quad \Gamma; \Delta \vdash P'}{\Gamma; \Delta \vdash \text{spawn } \{P\} P'} \quad \text{(REPIN)} \quad \frac{\Delta \vdash u : S \quad (\Gamma; \Delta) + \text{Env}(F) \vdash P \quad S \lesssim \langle \text{schof}(F) \rangle^I}{\Gamma; \Delta \vdash u?(F)P} \end{array}$$

for typing expressions to be sent as messages (see rule (OUT)). The second component is extended with the internal schema of the channel $\langle S \rangle^{I0}$; this definition is used for typing subjects of inputs and outputs (see rules (OUT), (SELECT), and (REPIN)).

Rule (MATCH) derives the typing of `match E with {Fi → Pii ∈ 1..n}` provided *E* and *P_i* are well typed in suitable environments. Additionally the rule enforces the exhaustiveness of the patterns – the hypothesis $S \lesssim \sum_{i \in 1..n} \text{schof}(F_i)$ – and their irredundancy – the hypotheses $(\text{schof}(F_j) \not\lesssim \sum_{j < i} \text{schof}(F_j))^{j \in 2..n}$ (our current compiler only warns the user in case of exhaustiveness and irredundancy are invalid). For example, if `x : a[] + b[]`, then

```
match x with
a[ ] -> P
```


is not exhaustive because it lacks a branch for the schema $b[]$. In

```
match x with
  a[] -> P
  b[] -> Q
  a[] + b[] -> R
```

the last branch is redundant. The presence of patterns whose schema is not a subschema of the expression is also checked. For example in

```
match x with
  a[] + int -> P
  b[] -> R
```

the type system point out that the first branch has a redundant sub-pattern.

5 Pattern matching and (local) operational semantics

This section defines the semantics of patterns and processes. In order to cope with values that may carry channels, both the pattern matching and the transition relation take an associated environment into account.

Let σ and σ' be two substitutions with disjoint domains. We write $\sigma + \sigma'$ to denote the substitution that is the union of σ and σ' . Every union in the following rules is always well defined because of the linearity constraint placed on patterns.

Definition 3 *The pattern match of a value V with respect to a pattern F in an environment Γ , written $\Gamma \vdash V \in F \rightsquigarrow \sigma$, is the least relation satisfying the following rules:*

$$\begin{array}{c}
\text{(VAL)} \\
\frac{\Gamma \vdash V : T \quad T \lesssim S}{\Gamma \vdash V \in S \rightsquigarrow \emptyset}
\end{array}
\quad
\begin{array}{c}
\text{(BIND)} \\
\frac{\Gamma \vdash V \in F \rightsquigarrow \sigma}{\Gamma \vdash V \in x : F \rightsquigarrow \{V/x\} + \sigma}
\end{array}$$

$$\begin{array}{c}
\text{(SEQ)} \\
\frac{a \in L \quad \Gamma \vdash V \in F \rightsquigarrow \sigma \quad \Gamma \vdash V' \in F' \rightsquigarrow \sigma'}{\Gamma \vdash a[V], V' \in L[F], F' \rightsquigarrow \sigma + \sigma'}
\end{array}$$

We write $\Gamma \vdash V \in F \not\rightsquigarrow$ when V does not match with F in Γ .

Rule (VAL) reduces pattern matching to the subschema relation between the schema of the value and the schema of the pattern (S). Rule (BIND) binds the value to the variable if the pattern matches the value. Rule (SEQ) accumulates

the result of the pattern matching of the components of a labelled schema. By linearity the substitutions σ and σ' have disjoint domains. For example

$$a[i : \text{int}], b[j : \text{Any}]$$

is a pattern that matches sequences made of an a -labelled integer followed by any b -labelled value. In case of success, i is bound to the integer, and j is bound to the content of the b -labelled value. For instance, the above pattern matches the value $a[5], b[c[\text{"hello"}]]$ yielding the substitution $\{5/i\}\{c[\text{"hello"}]/j\}$. Channel schemas occurring in patterns specify the schema of the channel literals to match. For example,

$$x : \langle a[] \rangle^0$$

matches every channel literal accepting values of schema $a[]$. It also matches every channel literal carrying values $a[] + b[]$, the reason being that matched channels can be used only for sending values and a channel carrying values of schema $a[] + b[]$ can safely be used for sending values of schema $a[]$.

The pattern-matching is computationally expensive since it uses the subschema relation. Such algorithm has a cost that is proportional to the product of the sizes of the pattern and of the received value, when the value does not contain channel literals. When the received value contains a channel literal the pattern matching amounts to computing the subschema relation between the schema of the channel and that of the pattern. While this computation is decidable because PiDuce schemas define *regular tree languages*, it has exponential cost in the size of the schemas [12]. If paying this cost at compile time may be acceptable, it becomes fateful at runtime because the performance degradation would be unacceptable. For instance an attacker might block a service by invoking it with channel literals, thus yielding a *denial of service*. Therefore in PiDuce we constraint schemas underneath channel constructors to be *labelled-determined* according to the next definition. This entails a subschema relation that can be verified in polynomial time [10].

Definition 4 Let $S \downarrow \ell$, read S has a labels ℓ , where ℓ is a set of labels, be the least relation such that:

$$\begin{array}{ll} () \downarrow \emptyset \\ \mathbf{T} \downarrow \emptyset \\ \langle S \rangle^\kappa \downarrow \emptyset \\ L[S], T \downarrow \widehat{L} & \text{if } \widehat{L} \neq \emptyset \text{ and there are } \ell, \ell' \text{ such that } S \downarrow \ell \text{ and } T \downarrow \ell' \\ S + T \downarrow \ell & \text{if } S \downarrow \ell', T \downarrow \ell'' \text{ and } \ell = \ell' \cup \ell'' \\ \mathbf{U} \downarrow \ell & \text{if } \mathcal{E}(\mathbf{U}) \downarrow \ell \end{array}$$

Schemas such that $S \downarrow$ is undefined are called empty.

The set ldet of labelled-determined schemas is the least set containing empty

schemas and such that:

- $() \in \text{ldet}$;
- $\langle S \rangle^\kappa \in \text{ldet}$, if $S \in \text{ldet}$;
- $L[S], T \in \text{ldet}$, if $S \in \text{ldet}$ and $T \in \text{ldet}$;
- $S + T \in \text{ldet}$, if $S \in \text{ldet}$, $T \in \text{ldet}$, and if $S \downarrow \ell$ and $T \downarrow \ell'$ then $\ell \cap \ell' = \emptyset$;
- $\mathbf{U} \in \text{ldet}$, if $\mathcal{E}(\mathbf{U}) \in \text{ldet}$.

We notice that $\perp \downarrow$ and $\mathbf{Empty} \downarrow$ are undefined, therefore \perp and \mathbf{Empty} are empty schemas. Similarly for $a[\], \mathbf{Empty}$. On the contrary, channel schemas always retains empty sets of labels. We also observe that $a[S] + (\sim \setminus a)[T]$ and $\sim[S] + \langle S \rangle^\kappa + \langle T \rangle^{\kappa'}$ are labelled-determined schemas. The schemas $a[\] + (a+b)[\]$ and $\langle a[\] + \sim[\] \rangle^\kappa$ are not labelled-determined. It is worth to emphasize that the labelled-determinedness restriction is applied to channel schemas only. For instance, $a[\] + \sim[\]$ is a legal **PiDuce** schema.

Next we define the (local) transition relation, namely the semantics of operations that are performed by a single **PiDuce** runtime environment. In particular we assume that subjects of selects and replications are local to the **PiDuce** runtime environment and every new channel is created locally – the distributed semantics is discussed in Section 6. To describe **URLs** of runtime environments, we let $1, 1', \dots$ range over a countably infinite set *locations*. We assume an injective relation $@$ mapping channel literals to locations and we write $u@1$ for u located at 1. With an abuse of notation, we will extend $-@1$ to variables. The relation $x@1$ is always true (since variables may be also instantiated by channel literals located at 1).

Let μ range over input labels $u?(F)$, bound output labels $(\Gamma)u!(V)$ with $\text{dom}(\Gamma) \subseteq \text{fn}(V)$, and τ . Let also $\text{fn}(u?(F)) = \{u\}$, $\text{fn}((\Gamma)u!(V)) = \{u\} \cup (\text{fn}(V) \setminus \text{dom}(\Gamma))$, $\text{bn}(u?(F)) = \text{fn}(F)$, $\text{bn}((\Gamma)u!(V)) = \text{dom}(\Gamma)$, and $\text{fn}(\tau) = \text{bn}(\tau) = \emptyset$.

Definition 5 *The (local) transition relation of **PiDuce**, $\Gamma \vdash_1 P \xrightarrow{\mu} Q$, is the least relation satisfying the rules in Table 4 plus the symmetric of rule (TR8) – the communication rule.*

It is worth to recall that we identify terms that are equal up-to alpha-conversion. In particular, if $\Gamma \vdash_1 P \xrightarrow{(\Gamma')u!(V)} Q$ then $\Gamma \vdash_1 P \xrightarrow{(\Gamma'\alpha)u!(V\alpha)} Q\alpha$ for every alpha-conversion α .

The above transition relation is similar to that of the π -calculus [34], except for the environment Γ . This environment is partially supplied by enclosing news and partially by the global environment. In particular, bound outputs gather an environment – see rule (TR4). This means that a communication between a sender and a receiver also carries information about the schema of

Table 4
Local transition relation in PiDuce.

$\frac{(TR1) \quad E \Downarrow V}{\Gamma \vdash_1 u!(E) \xrightarrow{u!(V)} \mathbf{0}}$	$\frac{(TR2) \quad (u_i @ 1)^{i \in I}}{\Gamma \vdash_1 \text{select } \{u_i?(F_i)P_i^{i \in I}\} \xrightarrow{u_i?(F_i)} P_i}$
$\frac{(TR3) \quad \Gamma + u:\langle S \rangle^\kappa \vdash_1 P \xrightarrow{\mu} Q \quad u \notin \text{fn}(\mu) \cup \text{bn}(\mu)}{\Gamma \vdash_1 \text{new } u:\langle S \rangle^\kappa \text{ in } P \xrightarrow{\mu} \text{new } u:\langle S \rangle^\kappa \text{ in } Q}$	
$\frac{(TR4) \quad \Gamma + v:\langle S \rangle^\kappa \vdash_1 P \xrightarrow{(\Gamma')u!(V)} Q \quad v \neq u \quad v \in \text{fn}(V) \setminus \text{dom}(\Gamma')}{\Gamma \vdash_1 \text{new } v:\langle S \rangle^\kappa \text{ in } P \xrightarrow{(\Gamma'+v:\langle S \rangle^\kappa)u!(V)} Q}$	
$\frac{(TR5) \quad E \Downarrow V \quad (\Gamma \vdash V \in F_i \nearrow)^{i \in 1..j-1} \quad \Gamma \vdash V \in F_j \leadsto \sigma}{\Gamma \vdash_1 \text{match } E \text{ with } \{F_i \rightarrow P_i^{i \in 1..n}\} \xrightarrow{\tau} P_j \sigma}$	
$\frac{(TR6) \quad \Gamma \vdash_1 P \xrightarrow{\mu} P' \quad \text{bn}(\mu) \cap \text{fn}(Q) = \emptyset}{\Gamma \vdash_1 \text{spawn } \{P\} Q \xrightarrow{\mu} \text{spawn } \{P'\} Q}$	$\frac{(TR7) \quad \Gamma \vdash_1 P \xrightarrow{\mu} P' \quad \text{bn}(\mu) \cap \text{fn}(Q) = \emptyset}{\Gamma \vdash_1 \text{spawn } \{Q\} P \xrightarrow{\mu} \text{spawn } \{Q\} P'}$
$\frac{(TR8) \quad \Gamma \vdash_1 P \xrightarrow{(\Gamma')u!(V)} P' \quad \Gamma \vdash_1 Q \xrightarrow{u?(F)} Q' \quad \text{dom}(\Gamma') \cap \text{fn}(Q) = \emptyset \quad \Gamma + \Gamma' \vdash V \in F \leadsto \sigma}{\Gamma \vdash_1 \text{spawn } \{P\} Q \xrightarrow{\tau} \text{new } \Gamma' \text{ in spawn } \{P'\} Q' \sigma}$	
$\frac{(TR9) \quad u @ 1}{\Gamma \vdash_1 u?*(F)P \xrightarrow{u?(F)} \text{spawn } \{P\} u?*(F)P}$	

the variables in the message. In practice this is the case: a Web service URL is always shipped with its WSDL document containing, for instance, the protocol that must be used to invoke the service and the schemas of arguments and of the result. This behaviour will be clear in the following, when we define the distributed PiDuce machine and its semantics. For the while we remark that the unique rules using Γ are (TR5) and (TR8) because of the pattern match in the premise.

We discuss rules (TR1), (TR5), and (TR8); the arguments about the other rules are omitted. The execution of $u!(E)$ amounts to evaluating E and, if this is possible, to delivering the value. The execution of $\text{match } E \text{ with } \{F_i \rightarrow P_i^{i \in 1..n}\}$ amounts to evaluating E and choosing the first pattern F_j matching with the value; the continuation P_j is run with the substitution returned by the pattern match algorithm. Rule (TR8) makes two parallel processes emitting and receiving a message on a same channel communicate. To this aim the message is matched against the pattern and the resulting substitution is applied to the receiver process. It is worth to notice that our semantics ad-

mits communications on variables (retaining a channel schema) that are not channel literals – *i.e.* they have no associated WSDL. Namely, as long as the communication remains local, the channel is not published. This publishing happens as soon as the channel is extruded to a remote machine, see rule (DTR1) in Section 6. In this respect, the current PiDuce prototype is less liberal: WSDLs are created as soon as the **new** is met in the object code.

6 Distributed operational semantics

The underlying model of PiDuce is distributed; it consists of a number of runtime environments – that may be PiDuce runtimes or not –, which execute at different locations and interact by exchanging messages over channels. In this section we describe the distributed semantics of our prototype implementation.

A PiDuce *machine* is a collection of runtime environments:

$$\Gamma_1 \vdash_{l_1} P_1 \parallel \dots \parallel \Gamma_n \vdash_{l_n} P_n$$

such that

- (1) l_1, \dots, l_n are pairwise different;
- (2) $\Gamma_1, \dots, \Gamma_n$ are *consistent* on commonly defined channel literals. Namely, if $u \in \text{dom}(\Gamma_i) \cap \text{dom}(\Gamma_j)$ then $\Gamma_i(u) = \Gamma_j(u)$;
- (3) $\Gamma_1, \dots, \Gamma_n$ are *localized* with respect to l_1, \dots, l_n , namely, for every u , if $u \in \text{dom}(\Gamma_i)$ and $u @ l_j$ then $u \in \text{dom}(\Gamma_j)$.

We let M, N , etc. range over PiDuce machines. We write $l \in \text{locn}(M)$ if M has a runtime environment at location l .

Processes in the runtime environments extend those defined in Table 1. They also include the following new operations that deal with remote locations:

- *input on remotely located channels*, namely in **select** $\{u_i?(F_i)P_i^{i \in I}\}$ are allowed subjects u_i that are not local;
- *new at remote location* **new** $u : \langle S \rangle^{I_0}$ **at** l **in** P that delegates remote channel managers to creating a channel literal at its own location;
- *import of remote PiDuce channel literals or one-way-interaction services* **import** $u : S = v$ **in** P that downloads the WSDL of the channel v , verifies that it is a subschema of S and replaces u with v in the continuation P .
- *import of remote request-response services* **import** $u : S$ **reply** $T = v$ **in** P that downloads the WSDL of the channel v , verifies that it is a subschema of $\langle \text{arg}[S], \langle T \rangle^0 \rangle^0$ and replaces u with v in the continuation P .

Table 5
Typing rules for distributed PiDuce.

$\frac{(\text{NEWAT}) \quad (\Gamma; \Delta) + u : \langle S \rangle^{\text{IO}} \vdash P}{\Gamma; \Delta \vdash \text{new } u : \langle S \rangle^{\text{IO}} \text{ at } l \text{ in } P}$	$\frac{(\text{IMPORT}) \quad (\Gamma; \Delta) + u : S \vdash P}{\Gamma; \Delta \vdash \text{import } u : S = v \text{ in } P}$
$\frac{(\text{IMPORT-RR}) \quad (\Gamma; \Delta) + u : \langle \text{arg}[S], \langle T \rangle^0 \rangle^0 \vdash P}{\Gamma; \Delta \vdash \text{import } u : S \text{ reply } T = v \text{ in } P}$	$\frac{(\text{LFORWD}) \quad \begin{array}{l} \Gamma \setminus \text{dom}(\Delta) \vdash u : S \quad \Gamma \vdash v : \langle T \rangle^0 \\ S \lesssim \langle T \rangle^{\text{I}} \end{array}}{\Gamma; \Delta \vdash u \multimap v}$

A further operation that regards remote locations is generated by the runtime environment:

- *linear forwarder* $u \multimap v$ that forwards a message on a channel u to v . These operators are used to implement inputs on remotely located channels; their theory has been developed in [18].

The type system of Table 3 is extended with the rules in Table 5 for news at remote locations, imports and linear forwarders. Rules (NEWAT), (IMPORT), and (IMPORT-RR) type remote news and imports checking P to be well typed in $(\Gamma; \Delta) + u : T$ (u is removed from Δ because u is a non local channel). Since the local process must be able to perform remote inputs over a channel created in a remote location, (NEWAT) requires a channel schema with input/output capability. Rule (LFORWD) types linear forwarders. The hypotheses require that u and v can be used for receiving and sending values, respectively; the judgment $\Gamma \setminus \text{dom}(\Delta) \vdash u : S$ verifies that u is not a local channel; the judgment $\Gamma \vdash v : \langle T \rangle^0$ verifies that v has been defined with output capability; $S \lesssim \langle T \rangle^{\text{I}}$ guarantees that values received on u can be safely forwarded to v .

Next we extend the (local) transition relation with the semantics of the operations dealing with remote locations. To this aim we drop the assumption in Section 5 that subjects of selects are local to the PiDuce runtime environment, as well as that new channels are always created locally to the runtime environment. With an abuse of notation we also use μ to range over labels $u : S$, $(u@1 : S)$, and $(\Gamma)u \multimap v$ with $\text{dom}(\Gamma) \subseteq \{v\}$. Let $\text{fn}(u@1 : S) = \{u\}$, $\text{fn}((u : S)) = \emptyset$, $\text{fn}((\Gamma)u \multimap v) = \{u, v\} \setminus \text{dom}(\Gamma)$ and let $\text{bn}(u : S) = \emptyset$, $\text{bn}((u@1 : S)) = \{u\}$, $\text{bn}((\Gamma)u \multimap v) = \text{dom}(\Gamma)$. We write $\text{spawn}_{i \in 1..n} \{P_i\}$ for $\text{spawn } \{P_1\} \cdots \text{spawn } \{P_n\}$. As usual \uplus denotes disjoint union.

Definition 6 *The transition relation $\Gamma \vdash_1 P \xrightarrow{\mu} Q$ and the distributed transition relation $\mathbf{M} \longrightarrow \mathbf{N}$ of PiDuce are the least relations satisfying the rules in Definition 5 plus:*

rules for $\Gamma \vdash_1 P \xrightarrow{\mu} Q$

$$\begin{array}{c}
\text{(TR10)} \\
\frac{(u_i @ 1)^{i \in I} \quad \left(u_j @ 1 \quad \Gamma \vdash u_j : \langle S_j \rangle^0 \right)^{j \in J}}{\Gamma \vdash_1 \text{select } \{ u_i ? (F_i) P_i^{i \in I \uplus J} \} \xrightarrow{\tau} \text{new } (v_j : \langle S_j \rangle^0)^{j \in J} \text{ in } \text{spawn}_{j \in J} \{ u_j \multimap v_j \} \text{select } \{ \quad u_i ? (F_i) (\text{spawn}_{k \in J} \{ v_k ? (x : S_k) u_k ! (x) \} P_i)^{i \in I} \quad v_j ? (F_j) (\text{spawn}_{k \in J \setminus j} \{ v_k ? (x : S_k) u_k ! (x) \} P_j)^{j \in J} \quad \}} \\
\text{(TR11)} \\
\frac{1 \neq 1'}{\Gamma \vdash_1 \text{new } u : \langle S \rangle^{10} \text{ at } 1' \text{ in } P \xrightarrow{(u @ 1' : \langle S \rangle^{10})} P} \quad \text{(TR12)} \quad \Gamma \vdash_1 \text{import } u : S = v \text{ in } P \xrightarrow{v : S} P\{v/u\} \\
\text{(TR13)} \quad \Gamma \vdash_1 \text{import } u : S \text{ reply } T = v \text{ in } P \xrightarrow{v : \langle \text{arg}[S] \rangle, \langle T \rangle^0} P\{v/u\} \quad \text{(TR14)} \quad \Gamma \vdash_1 u \multimap v \xrightarrow{u \multimap v} \mathbf{0} \\
\text{(TR15)} \quad \frac{\Gamma + v : \langle S \rangle^\kappa \vdash_1 P \xrightarrow{u \multimap v} Q \quad w @ 1 \quad w \notin \text{dom}(\Gamma) \cup \text{fn}(P)}{\Gamma \vdash_1 \text{new } v : \langle S \rangle^\kappa \text{ in } P \xrightarrow{(w : \langle S \rangle^\kappa) u \multimap w} Q\{w/v\}} \\
\text{(TR16)} \quad \frac{\Gamma \vdash_1 P \xrightarrow{u : S} Q \quad u @ 1 \quad \Gamma \vdash u : \langle S' \rangle^\kappa \quad \langle S' \rangle^\kappa \preceq S}{\Gamma \vdash_1 P \longrightarrow Q}
\end{array}$$

rules for $M \longrightarrow N$

$$\begin{array}{c}
\text{(DTR1)} \\
\frac{\Gamma \vdash_1 P \xrightarrow{(v_i : S_i^{i \in I}) u ! (V)} Q \quad u @ 1' \quad (w_i @ 1 \quad w_i \notin \text{dom}(\Gamma) \cup \text{fn}(P))^{i \in I} \quad \Gamma'' = \Gamma|_{\text{fn}(V) \setminus \{v_i^{i \in I}\}}}{\Gamma \vdash_1 P \parallel \Gamma' \vdash_{1'} R \longrightarrow \Gamma + w_i : S_i^{i \in I} \vdash_1 Q\{w_i/v_i^{i \in I}\} \parallel \Gamma' + w_i : S_i^{i \in I} + \Gamma'' \vdash_{1'} \text{spawn } \{u!(V\{w_i/v_i^{i \in I}\})\} R} \\
\text{(DTR2)} \\
\frac{\Gamma \vdash_1 P \xrightarrow{(u @ 1' : \langle S \rangle^{10})} Q \quad v @ 1' \quad v \notin \text{dom}(\Gamma')}{\Gamma \vdash_1 P \parallel \Gamma' \vdash_{1'} R \longrightarrow \Gamma + v : \langle S \rangle^{10} \vdash_1 Q\{v/u\} \parallel \Gamma' + v : \langle S \rangle^{10} \vdash_{1'} R} \\
\text{(DTR3)} \\
\frac{\Gamma \vdash_1 P \xrightarrow{u : S} Q \quad u @ 1' \quad \Gamma' \vdash u : \langle S' \rangle^\kappa \quad \langle S' \rangle^\kappa \preceq S}{\Gamma \vdash_1 P \parallel \Gamma' \vdash_{1'} R \longrightarrow \Gamma + u : \langle S' \rangle^\kappa \vdash_1 Q \parallel \Gamma' \vdash_{1'} R} \\
\text{(DTR4)} \\
\frac{\Gamma \vdash_1 P \xrightarrow{(\Gamma'') u \multimap v} Q \quad u @ 1' \quad \Gamma' \vdash u : \langle S \rangle^\kappa \quad \text{dom}(\Gamma'') \cap \text{dom}(\Gamma') = \emptyset \quad \Gamma''' = \Gamma|_{\{v\}} + \Gamma''}{\Gamma \vdash_1 P \parallel \Gamma' \vdash_{1'} R \longrightarrow \Gamma + \Gamma'' \vdash_1 Q \parallel \Gamma' + \Gamma''' \vdash_{1'} \text{spawn } \{u?(x : S) v!(x)\} R} \\
\text{(DTR5)} \quad \frac{\Gamma \vdash_1 P \xrightarrow{\tau} Q}{\Gamma \vdash_1 P \longrightarrow \Gamma \vdash_1 Q} \quad \text{(DTR6)} \quad \frac{M \longrightarrow M'}{M \parallel N \longrightarrow M' \parallel N}
\end{array}$$

Rule (TR10) defines selects with remote subjects. It translates the select process on-the-fly into another one using a local select. (This translation has been proposed for encoding distributed choice in [18].) We analyze the case of a select with three branches, one with a local subject u and the others with remote subjects v and w :

$$\text{select } \{u?(F)P \quad v?(F')Q \quad w?(F'')R\}$$

This select may be turned into a local select by creating two local siblings for v and w , let them be v' and w' , respectively, and communicating to the channel managers of v and w the presence of these siblings. So the above process might be translated into

$$\text{new } v', w' : S', T' \text{ in } \text{spawn } \{v \multimap v'\} \text{ spawn } \{w \multimap w'\} \\ \text{select } \{u?(F)P \quad v'?(F')Q \quad w'?(F'')R\}$$

But this translation is too rude because of the following problem. The purpose of the linear forwarder $v \multimap v'$ is to migrate to the remote location of v and forward one message to the location of v' – the same of u . Similarly for $w \multimap w'$. Now, by the semantics of select in Definition 5, the branch $u?(F)P$ may be chosen because of the presence of a message on u . This choice destroys the branches $v'?(F')Q$ and $w'?(F'')R$. Therefore, when messages for v' and w' will be delivered by the remote machines, such messages will never be consumed. To avoid such misbehaviour, one has to compensate the previous emission of linear forwarders by undoing them with $v'?(x : S') \ v!(x)$ and $w'?(x : T') \ w!(x)$. In case the picked branch is $v'?(F')Q$, by a similar argument, we have to compensate only one linear forwarder – the $w \multimap w'$. Therefore the correct translation for the distributed select is

$$\text{new } v', w' : S', T' \text{ in } \text{spawn } \{v \multimap v'\} \text{ spawn } \{w \multimap w'\} \\ \text{select } \{ u?(F)(\text{spawn } \{v'?(x : S') \ v!(x)\} \\ \text{spawn } \{w'?(x : T') \ w!(x)\} \ P) \\ v'?(F')(\text{spawn } \{w'?(x : T') \ w!(x)\} \ Q) \\ w'?(F'')(\text{spawn } \{v'?(x : S') \ v!(x)\} \ R) \}$$

that is the term yielded by the rule in this case.

Rule (TR11) creates a channel remotely located at l' . To this aim a channel literal located at l' is taken and the local name is replaced by this channel literal in the continuation. When $l = l'$, the process $\text{new } u : \langle S \rangle^{I0} \text{ at } l' \text{ in } P$ is syntactic sugar for $\text{new } u : \langle S \rangle^{I0} \text{ in } P$. In this case its semantics is defined by rules (TR3) and (TR4). Rule (DTR2) guarantees that such a literal is fresh at the remote location. Rule (TR12) imports a channel literal v casting its schema to $\langle S \rangle^\kappa$ (the compiler type-checks the continuation under the assumption $u : \langle S \rangle^\kappa$ – see (IMPORT)). Rules (TR16) and (DTR3) checks the consistency of this cast at runtime. In particular they verify that the true schema of v is a subschema of $\langle S \rangle^\kappa$; in case of failure a runtime type error is raised. Rule

(TR13) defines the import of request-response services in a way similar to (TR12). Rule (TR14) defines a linear forwarder by lifting to the label the linear forwarder itself. This rule and rule (DTR4) define a linear forwarder $u \multimap v$ as a small atom migrating to the remote location of u and becoming the process $u?(x : S)v!(x)$. It is also possible that v is a channel literal local to the sender – see rule (TR15). In this case the environment of the receiver must be extended adequately.

Rule (DTR1) models the delivery of a message to a remote runtime environment $1'$. When this occur all the bound channels are created in the sender location 1 . The message is put in parallel with every process running at $1'$. The other rules have been already described, except (DTR5) and (DTR6), that lift transitions in components to composites machines.

We conclude this section by asserting the soundness of the static semantics. Proofs are reported in the Appendix A. The first property, subject reduction, states that well-typed processes always transit to well-typed processes. Let $[\Gamma]_1^{I0}$ be the environment

$$[\Gamma]_1^{I0}(u) = \begin{cases} \langle S \rangle^{I0} & \text{if } u@1 \text{ and } \Gamma(u) = \langle S \rangle^\kappa \\ \text{undefined} & \text{otherwise} \end{cases}$$

The environment Γ contains all the bindings of channel literals and variables when used as values. However, channels local at 1 may be used in the runtime 1 with input and output capability (*cf.* rule (NEW) in Table 3). The operation $[\Gamma]_1^{I0}$ is meant to define the environment for local channels: it extracts the channels local at 1 out of Γ and replaces the capability with $I0$. We recall that, according to our notation, if $x \in \text{dom}(\Gamma)$ with channel schema then $x \in \text{dom}([\Gamma]_1^{I0})$, too, because $x@1$ is always true.

Theorem 7 (*Subject Reduction*) *Let $\Gamma; [\Gamma]_1^{I0} \vdash P$. Then*

- (1) *if $\Gamma \vdash_1 P \xrightarrow{(\Gamma')u!(V)} Q$ then (a) $\Gamma + \Gamma'; [\Gamma + \Gamma']_1^{I0} \vdash Q$, (b) $\Gamma + [\Gamma]_1^{I0} \vdash u : S$, $\Gamma + \Gamma' \vdash V : T$ and $S \lesssim \langle T \rangle^0$;*
- (2) *if $\Gamma \vdash_1 P \xrightarrow{u?(F)} Q$ then (a) $(\Gamma; [\Gamma]_1^{I0}) + \text{Env}(F) \vdash Q$ and (b) $\Gamma + [\Gamma]_1^{I0} \vdash u : S$ with $S \lesssim \langle \text{schof}(F) \rangle^I$;*
- (3) *if $\Gamma \vdash_1 P \xrightarrow{(\Gamma')u \multimap v} Q$ then (a) $\Gamma + \Gamma'; [\Gamma + \Gamma']_1^{I0} \vdash Q$ and (b) $\Gamma \setminus \text{dom}([\Gamma + \Gamma']_1^{I0}) \vdash u : S$, $\Gamma + \Gamma' \vdash v : \langle T \rangle^0$ and $S \lesssim \langle T \rangle^I$;*
- (4) *if $\Gamma \vdash_1 P \xrightarrow{(u@1':\langle S \rangle^{I0})} Q$ then $(\Gamma; [\Gamma]_1^{I0}) + u : \langle S \rangle^{I0} \vdash Q$;*
- (5) *if $\Gamma \vdash_1 P \xrightarrow{v:S} Q$ then (a) $\Gamma \vdash v : \langle T \rangle^\kappa$ and (b) $(\Gamma; [\Gamma]_1^{I0}) + v : \langle T \rangle^\kappa \vdash Q$ with $\langle T \rangle^\kappa \lesssim S$;*
- (6) *if $\Gamma \vdash_1 P \xrightarrow{\tau} Q$ then $\Gamma; [\Gamma]_1^{I0} \vdash Q$.*

Let $\vdash M$ if, for every $\Gamma \vdash_1 P$ in M : $\Gamma; [\Gamma]_1^{I0} \vdash P$. Then

(7) if $\vdash M$ and $M \longrightarrow N$ then $\vdash N$.

The first item of the subject reduction entails that the reduct Q of a $(\Gamma')u!(V)$ -transition is typable provided the initial process P is typable. To this aim, the environment $\Gamma; [\Gamma]_1^{I0}$ must be suitably extended with the bindings in Γ' . This extension is similar to the one used in the rule (NEW) of the type system. In facts, bindings in Γ' are collected by surrounding news – see rule (TR4). The seventh item regards machines. We say that a machine is well-typed if every runtime environment $\Gamma \vdash_1 P$ is well-typed.

The second soundness property concerns *progress*, that is, an output on a channel will be consumed if an input on the same channel is available. In order to guarantee progress, it is necessary to restrict (well-formed) environments. To illustrate the problem, consider the following judgment:

$$u : \langle \text{int} + \text{string} \rangle^\kappa, v : \text{int} + \text{string} \vdash_1$$

$$\text{spawn } \{u!(v)\} \quad u?(x : \text{Any}) \text{ match } x \text{ with } \{\text{int} \rightarrow P \quad \text{string} \rightarrow Q\}$$

The reader may verify that it is derived by our type system, however the pattern matching will fail because the schema of v is neither a subschema of **int** or of **string**. Another example is the following. Let Γ be $u : a[b[]]$, $V = u$, and $F = a[v : b[]]$. Then $\Gamma \vdash V : S$ and $S \lesssim \text{schof}(F)$ but there is no σ such that $\Gamma \vdash V \in F \rightsquigarrow \sigma$. In fact these circumstances never occur in practice: if a value is sent, it may contain either constants or channel literals. Under this constraint, progress is always guaranteed.

We say that Γ is *channeled* if, for every $u \in \text{dom}(\Gamma)$, $\Gamma(u)$ is a channel schema.

Proposition 8 *Let Γ be channeled, $\Gamma \vdash V : S$, and $S \lesssim \text{schof}(F)$. Then there is σ such that $\Gamma \vdash V \in F \rightsquigarrow \sigma$.*

Theorem 9 (Progress) *Let Γ be channeled and let $\Gamma; [\Gamma]_1^{I0} \vdash P$. If $\Gamma \vdash_1 P \xrightarrow{(\Gamma')u!(V)} Q'$ and $\Gamma \vdash_1 P \xrightarrow{u?(F)} Q''$ then there is Q such that $\Gamma \vdash_1 P \xrightarrow{\tau} Q$.*

The progress may be also established for machines. For instance it is possible to demonstrate that if $\Gamma; [\Gamma]_1^{I0} \vdash P$, $\Gamma \vdash_1 P \xrightarrow{(u@1':(S)^\kappa)} Q$, and $1' \in \text{locn}(\mathbf{M})$ then $\mathbf{M} \parallel \Gamma \vdash_1 P \longrightarrow \mathbf{N}$, for some \mathbf{N} . We omit the formal statements of these properties; the corresponding proofs are immediate.

7 The PiDuce runtime environment

PiDuce runtime environments consist of two components: the *virtual machine* and the *channel manager* – see Figure 1.

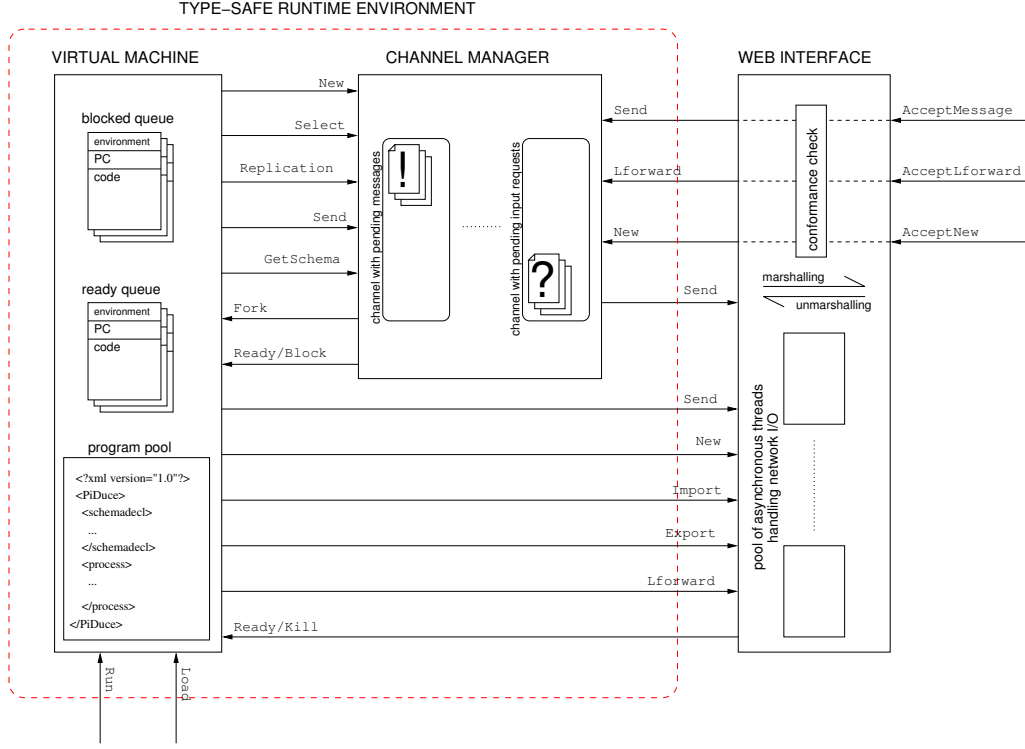


Fig. 1. PiDuce: the runtime.

The *virtual machine* executes threads by interpreting PiDuce object code. Threads arise from processes that have been explicitly added to the scheduler and from processes spawned as a consequence of another process execution. The *channel manager* handles the pool of channels literals that are local to the runtime environment. It is thus responsible for any operation involving local channels, in particular creation, send, and receive operations. The PiDuce runtime environment interacts with the external environment by means of a *Web interface* – see Figure 1 and Section 8 –, a firewall implementing mechanisms for adhering to particular protocols and for publishing PiDuce services to the outer world using standard technologies. The Application Programming Interface provided by the components of the PiDuce architecture is summarized in Table 6. The next sections give a detailed description of the most important operations.

The modular design of this architecture has four main consequences: (1) the channel manager and the Web interface may be used stand-alone for providing PiDuce-compatible communication primitives in (native) programs written in a language different from PiDuce; (2) the virtual machine and the channel manager are decoupled from the actual transport protocols and technologies used in distributed communication. In this way a large part of the prototype may be adapted to different contexts with minimum effort; (3) communications occurring within the same runtime environment are short-circuited and do not entail any additional overhead because they solely rely on internal

Table 6
Application Programming Interface of PiDuce.

Method	Arguments	Description
Virtual machine		
Load	XmlDocument code	loads a PiDuce process a reference to the root element of the compiled PiDuce process to load
Fork	XmlNode PC Env env	creates a new thread first instruction of the thread to execute initial environment
Run		executes the scheduler
Channel manager		
New	Schema schema Capability cap	creates a local channel schema of the values transported by the channel exported capability of the channel
Select	Channel[] channels Pattern[] patterns XmlNode[] PCs Thread thread	performs a <code>select</code> input operation array of subjects array of patterns array of continuations caller thread
Replication	Channel channel Pattern pattern XmlNode PC Env env	performs a replication operation subject pattern continuation initial environment of the threads that will be spawned
Lforward	Channel lchannel URL rchannel	performs a linear forwarding operation local channel remote channel literal
Output	Channel channel Value value	sends a value to a local channel destination channel value to send
GetSchema	Channel channel	retrieves the schema of a channel channel
WEB interface		
IsLocal	URL channel	verifies whether a channel is local or remote channel literal to verify
Output	URL channel Value value	sends a value to a remote channel remote channel literal value to send
New	Schema schema Capability cap URL location Thread thread int index	creates a remote channel schema of the values transported by the channel exported capability of the channel location of the remote channel manager thread to wake up upon completion index of <code>thread</code> 's environment to update
Import	Schema schema URL rchannel Thread thread int index	imports an existing remote channel schema of the imported channel remote channel literal thread to wake up upon completion index of <code>thread</code> 's environment to update
Export	Channel channel	publishes a local channel local channel to publish
Lforward	URL rchannel Channel channel	sends a linear forwarder remote source channel literal local destination channel
WEB interface (asynchronous methods)		
AcceptMessage	Channel channel XmlNode value	accepts an incoming message for a local channel destination channel marshalled message
AcceptNew	Schema schema Capability cap	accepts the creation of a local channel schema of the values transported by the channel exported capability of the channel
AcceptLforward	Channel channel URL rchannel	accepts a linear forwarder local source channel remote destination channel

data structures, rather than passing through the Web interface; (4) the virtual machine and the channel manager realize a type-safe environment: every operation performed therein can never manifest a type error.

7.1 *The virtual machine*

The virtual machine executes compiled **PiDuce** processes by means of a round-robin scheduler. The virtual machine stores its data in three structures: the *program pool*, containing the object code of the processes that have been loaded; the *ready queue*, containing threads that are ready to execute; the *blocked queue*, containing threads awaiting for some message.

Every thread retains a *program counter* and an *environment*. The former is a pointer to the next instruction to be executed; the latter stores the associations between variables and values (in the **PiDuce** object code variables have been translated into indexes of the environment). The virtual machine exposes the operations of object code loading, of thread creation, and of scheduling that are detailed below.

Object code loading is implemented by `Load(XmlDocument code)` that adds a compiled **PiDuce** process pointed by `code` to the program pool and schedules its main thread for execution. The environment of the new thread is empty.

Thread creation is implemented by `Fork(XmlNode PC, Env env)` that forks a new thread retaining a program counter `PC` and an environment `env`. This operation is a consequence of a `spawn` or a replication instruction.

Scheduling is implemented by `Run()`. The scheduler selects the first thread from the ready queue and executes it for a fixed number of instructions or until it blocks. After execution the thread is enqueued at the end of the ready queue if it has been preempted or at the end of the blocked queue if it has been blocked awaiting for a message. This round-robin policy, in combination with the FIFO policy of the channel manager queues, guarantees fair execution of threads. The execution of **PiDuce** object code instructions is detailed below. Overall, those instructions that address local channels are managed by invocations to methods of the channel manager; those that address interactions with the external environment are managed by invocations to methods of the Web interface.

Output to local channels amounts to evaluating the expression to a **value** and invoking the channel manager with `Output(chan, value)`. Select is evaluated according to whether the corresponding subjects are local or not. This is verified by comparing the channel literals and the URL of the local Web interface. If the subjects are all local, the branches of the select are organized in three arrays `channels`, `patterns`, and `PCs` whose i -th entry respectively represents the i -th subject, the i -th pattern and the i -th continuation. Then the channel manager is invoked with `Select(channels, patterns, PCs, thread)`, where `thread` is the caller thread. This operation returns a boolean; if this value is **true** – a branch has been chosen –, `thread` is moved to the ready queue, otherwise it is moved to the blocked queue – no branch has been chosen. If the subjects are also remote then the code is rewritten according to the rule (TR10) in Definition 6. New at the local location amounts to invoking the channel manager with `New(schema, capability)`; this method returns a reference `lchannel` to the new channel, which is used to extend the environment of the caller and to invoke the method `Export(lchannel)` of the Web interface. Match and spawn are computed internally by the virtual machine (their description is omitted). Replication has always one subject that is local; in this case the channel manager is invoked with `Replication(channel, pattern, PC, env)`, where `env` is the environment of the caller; the thread of the caller is terminated. Import of local channels amounts to invoking the channel manager with `GetSchema(channel)` that returns the schema of `channel`; hence the subschema verification is undertaken and the thread progresses or terminates with a type-exception according to the result of the verification.

Instructions that address channels at remote locations are executed as follows. Output to a remote channel amounts to evaluating the expression to a **value** and to invoking the Web interface with `Output(chan, value)`. New at a remote location invokes the Web interface with `New(schema, location, thread)` and moves `thread` to the blocked queue. Import of channels at remote locations invokes the Web interface with `Import(schema, location, thread)` and moves `thread` to the blocked queue. Linear forwarder is implemented by invoking `Lforward(rchannel, lchannel)` of the Web interface.

7.2 The channel manager

The channel manager handles the pool of channels that are locally defined in the PiDuce runtime environment. It implements channel creation and the related input/output operations. A channel consists of a *schema*, describing the values that are carried, a *capability* determining the permitted operations on the channel (input, output, or input/output), a *message queue* containing all the messages that have been sent but not consumed, and a *request queue* containing the *input requests* of threads waiting for a message on that channel.

Whenever a new message arrives, the first input request, if any, is *notified*; otherwise the message is enqueued. (Because of exhaustiveness of inputs, every channel has always at most one queue that is nonempty.) Input requests are classified as follows:

- **SelectRequest**(*thread*, *pattern*, *PC*, *next*) corresponds to a select operation. An instance of this object is inserted in the request queue of every subject of a **select**, and all the instances are linked together in a circular list through the *next* field, indicating that they are related. When this request is notified with a value *v*, *thread* is unblocked, its program counter is set to *PC* – the continuation corresponding to the branch that has been activated –, *thread*'s environment is updated with the bindings yielded by the matching of the received value with *pattern*, and *all* the instances of **SelectRequests** reachable by the *next* field are removed from the corresponding request queues;
- **ReplicationRequest**(*pattern*, *PC*, *env*) corresponds to a replication operation. When this request is notified with a value *v* a new thread is forked with the method **Fork**(*PC*, *env'*) of the virtual machine where *env'* is *env* updated with the bindings yielded by matching *v* with *pattern*. Finally, the same request is enqueued again;
- **LforwardRequest**(*rchannel*) corresponds to a linear forwarder created by a remote PiDuce runtime environment. When this request is notified with a value *v*, the method **Output**(*rchannel*, *v*) of the Web interface is invoked in order to forward *v* to *rchannel*. The request is then removed.

The operations defined by the channel manager are detailed below.

Channel creation is implemented by **New**(**Schema** *schema*, **Capability** *cap*). This method defines and initializes the message and the request queues of the new channel and returns a reference to it.

Channel input is implemented by several methods. The **Select**(**Channel** [] *channels*, **Pattern** [] *patterns*, **XmlNode** [] *PCs*, **Thread** *thread*) method inspects the message queues of *channels* one by one. If there is a value in one of them, let it be the *i*-th channel, this value is matched against *patterns*[*i*]. Let σ be the substitution returned by the pattern matching algorithm and let *env* be the environment of *thread*. Then *env* is extended with σ and *thread*'s program counter is set to *PCs*[*i*]. In this case **select** returns **true** (*thread* is not blocked). If there is no message, a **SelectRequest** object is created for every element in *channels*. These objects are all linked together. In this case **select** returns **false** (*thread* is moved to the blocked queue).

The method **Replication**(**Channel** *channel*, **Pattern** *pattern*, **XmlNode**

`PC`, `Env env`) inspects the message queue of `channel`. If there is no value, it creates a `ReplicationRequest` object and enqueues it in `channel`'s request queue. If the message queue is nonempty, it spawns as many threads as the values in the message queue. Let V_1, \dots, V_n be such values. The spawned threads have program counters set to `PC` and environments set to `env` extended with the substitution returned by the matching of V_i against `pattern`. Finally a `ReplicationRequest` object is created and enqueued in `channel`'s request queue.

The method `Lforward(Channel lchannel, URL rchannel)` implements the spawn described by rule (DTR1) in Definition 6 as follows. If the message queue of `lchannel` is nonempty, the first value, let it be `value`, is dequeued and the method `Output(rchannel, value)` of the Web interface is invoked. Otherwise a `LforwardRequest` object is created and enqueued in `channel`'s request queue.

Channel output is implemented by the method `Output(Channel channel, Value value)` that sends `value` on `channel`. If `channel`'s request queue is nonempty, the first input request is notified with `value`. Otherwise `value` is enqueued in the message queue.

8 The Web interface

The Web interface takes care of any task whose fulfillment implies accessing the network. This is a problematic component of the `PiDuce` architecture since it ultimately makes `PiDuce` interoperable with the available technologies. In this section we detail the operations provided by the Web interface, the mechanisms interfacing `PiDuce` channels and Web services, and the correspondence of `PiDuce` schemas with XML-Schema.

8.1 Operations of the Web interface

There are two groups of operations: those – `Output`, `New`, `Import`, `Export` and `Lforward` – that are invoked by the local runtime environment and those – `AcceptMessage`, `AcceptNew`, and `AcceptLforward` – that are invoked by the network. In any case, the operations that involve potentially blocking network actions are executed asynchronously with respect to the rest of the code. In this way the Web interface is always ready for handling requests, thus guaranteeing liveness. The `Output` and `AcceptMessage` operations must respectively

translate PiDuce values into XML documents – *marshalling* – and XML documents into PiDuce values – *unmarshalling*. The operation of marshalling is trivial except that it loses type information of constant values. For example, the values `a["5"]` and `a[5]` are both marshalled into `<a>5`. Henceforth the operation of unmarshalling is not merely the inverse of marshalling. Indeed this operation uses the schema of the channel to infer the right constants. In case of ambiguity, for instance when the schema is `a[int + string]`, the returned value is chosen among the possible ones.

Each operation is discussed below.

Remote channel output is implemented by `Output(URL rchannel, Value value)`. This method sends the value `value` to `rchannel`. The Web interface behaves differently according to the interaction pattern of the service, which is found in the WSDL of `rchannel`. In the case of *one-way interaction pattern* the method marshals `value` into an XML document, and sends the marshalled value to `rchannel`. In the case of *request-response interaction pattern* the value sent must have the shape `arg[V], r`, where `r` is a channel for receiving the message returned by `rchannel`. The Web interface extracts `V` from this value, opens a connection with `rchannel`, sends the marshalled `V`, and waits for a response message *on the same connection*. When the response message is received, the Web interface sends it on `r` after verifying that the message conforms with schema of `r`.

Request of channel creation is implemented by `New(Schema schema, Capability cap, URL location, Thread thread, int i)`. This method encodes `schema` and `cap` into an XML document and sends it to `location`. The encoding is discussed in Section 8.3. The PiDuce machine at `location` replies with the URL of the WSDL of the fresh channel literal that has been created. This URL is used to set the `i`-th entry of `thread`'s environment. Finally, `thread` is moved back into the ready queue.

Importation of a remote channel is implemented by `Import(Schema schema, URL rchannel, Thread thread, int i)`. This method downloads the schema of `rchannel` by accessing its WSDL and verifies that `schema` is a subschema of the downloaded schema. If this is the case, the `i`-th entry of `thread`'s environment is set to the imported channel literal and `thread` is moved into the ready queue. Otherwise, `thread` is terminated.

Exportation of a remote channel is implemented by `Export(Channel lchannel)`. This method creates the WSDL resource for `lchannel` and makes

it available over the Web at a fresh URL.

Outgoing linear forwarder is implemented by `Lforward(URL rchannel, Channel lchannel)`. This method sends the URL of the WSDL associated with `lchannel` to the PiDuce runtime environment owning `rchannel` (`lchannel` must have been previously exported).

Message delivery is implemented by `AcceptMessage(Channel lchannel, XmlNode value)`. This method is asynchronous; it is invoked when there is a message from the network whose destination is the URL corresponding to the local channel `lchannel`. The Web interface unmarshals `value` into an actual PiDuce value, verifies that `value` conforms with the schema of `lchannel`, and invokes the method `Output(lchannel, value)` of the channel manager. Conformance amounts to verifying that the channel has been created with either output or input/output capability and that the schema of `value` is a subschema of the schema carried by `lchannel`. If one of these verifications fails the value is discarded and a type error message is sent back on the same connection.

Channel creation is implemented by `AcceptNew(Schema schema, Capability cap)`. This method is asynchronous; it is invoked when the network delivers a remote request for creating a local channel with schema `schema` and exported capability `cap`. The Web interface first invokes `New(schema, cap)` of the local channel manager, then exports it to the Web invoking `Export(lchannel)` where `lchannel` is the reference to the new channel returned by `New`. This last invocation returns a fresh channel literal – the URL of the WSDL resource – that is sent back on the same connection.

Incoming linear forwarder is implemented by `AcceptLforward(Channel lchannel, URL rchannel)`. This method is asynchronous; it is invoked when the network delivers a linear forwarder whose destination is the URL corresponding to `lchannel`. The Web interface verifies that the channel has been exported with either input or input/output capability and invokes `Lforward(lchannel, rchannel)` of the channel manager. If the capability verification fails an error is sent back on the same connection.

8.2 Mechanisms interfacing PiDuce channels and Web services

Web services are published by interfaces that are written in a standard format: the WSDL – Web Service Description Language [29]. Every WSDL interface contains two parts: the *abstract one* defining the set of operations supported by the service, and the *concrete one* binding every operation to a concrete network protocol and to a concrete location. Every operation is described by a name and by the schema of the *messages* that the operations process and/or produce. Albeit WSDL does not make any commitment on the schema language to be used, XML-Schema is the schema language universally adopted. Operations have an associated interaction pattern that conforms to one out of four models: *one-way interaction* (the client invokes a service by sending a message); *notification* (the service sends the message); *request-response* (the client sends a message and waits for the response); *solicit-response* (the service makes a request and waits for the response). Because of asynchrony, PiDuce channel literals conform with the one-way interaction pattern. Therefore it is easy to interface PiDuce channels and one-way Web services. The extension to other interaction patterns is in any case relevant; we discuss such extension in Section 10.

We discuss the possible WSDL interfaces by analyzing a number of examples. Consider the process $\text{new } u : \langle S \rangle^\kappa$ in P . This process creates a channel literal u and publishes it in a WSDL interface whose abstract part is:

```
<schema>
  <complexType name="InSchema">⌈ S ⌋</complexType>
</schema>
<message name="input">
  <part name="par" type="InSchema"/>
</message>
<portType name="service">
  <operation name="operation">
    <input message="Input"/>
  </operation>
</portType>
```

($\llbracket S \rrbracket$ is the XML-Schema encoding of the PiDuce schema S defined in Section 8.3.) This operation, being one-way, defines the "input" message only and its schema "Input". The concrete part of the WSDL interface for x is specified by two elements, **binding** and **service**:

```
1 <binding name="serviceSoap" type="service">
2   <soap:binding transport="http://schemas.xmlsoap.org/soap/http"/>
3   <operation name="operation">
```

```

4     <soap:operation style="document"
5         soapAction="http://www.cs.unibo.it:1811/x"/>
6     <input><soap:body use="literal"/></input>
7 </operation>
8 </binding>

```

The element **binding** defines the concrete message formats and the protocols to be used for accessing the operation. Currently, PiDuce only supports the SOAP-over-HTTP binding – see line 2 of the above document. This means that PiDuce Web interfaces communicate SOAP messages (XML documents with the shape `Envelope[Header[headers], Body[parameters]]` where the **Header** is optional) using the HTTP protocol. The **soap:operation** element on line 4 has two attributes: **style** specifies that the operation style is **document**; **soapAction** specifies the SOAPAction header used in the HTTP request. The information in these two attributes indicates that the transported XML message appears directly under the SOAP Body element without any additional encoding. Therefore a possible SOAP message for invoking a service having schema $\langle a[\text{int}] + b[\text{string}] \rangle^0$ is

```

<?xml version="1.0" encoding="utf-8"?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
    <env:Body>
        <a>1</a>
    </env:Body>
</env:Envelope>

```

The element **service** connects a binding to a specific URL. This URL is given by the location of the runtime environment followed by a unique path, which is typically formed by appending **?wsdl** suffix to the name of the channel. For instance the following **service** element asserts that the service is located at `http://www.cs.unibo.it:1811/u` and that it is published with IO capability (line 3):

```

1 <service name="service">
2     <port name="service" binding="serviceSoap">
3         <soap:address location="http://www.cs.unibo.it:1811/u"
4             piduce:operationCapability="IO">
5         </soap:address>
6     </port>
7 </service>

```

It is worth to notice the use of the nonstandard attribute **piduce:operationCapability** (line 4) which notifies PiDuce clients that the service may support remote inputs. As the attribute is in the **piduce** namespace, it will be ignored by standard Web services.

Apart defining new channels, PiDuce also permits to import externally defined services. The process `import $u:S = \text{URL}$ in P` imports a one-way interaction service located at `URL` and gives it the name u . The XML-Schema of the service u is extracted from the WSDL located at `URL`, it is decoded into a PiDuce schema T , and the decoded schema is verified to be compatible with S . If the attribute `piduce:operationCapability="κ"` is found in the WSDL (implying that u has been published by a PiDuce runtime), compatibility means $S \lesssim \langle T \rangle^\kappa$. Otherwise compatibility means $S \lesssim \langle T \rangle^0$. The Web interface also verifies whether the binding is SOAP over HTTP. In case of success the value of the attribute `location` in the `service` element is used as target for future invocations. In case of failure of any of the above checks, the continuation P is not executed.

When the externally defined service is request-response, it may be imported by `import $u:S$ reply $T = \text{URL}$ in P` . The schema of u is retrieved as before but, in this case, the WSDL interface has a `portType` element whose shape is:

```
<portType name="op-request-response">
  <operation name="request-response">
    <input message="Input"/>
    <output message="Output"/>
  </operation>
</portType>
```

The Web interface decodes `Input` and `Output` into the schemas S_I and S_O , respectively. Then it verifies that $\langle \text{arg}[S], \langle T \rangle^0 \rangle^0 \lesssim \langle \text{arg}[S_I], \langle S_O \rangle^0 \rangle^0$. The remaining behaviour is similar to the previous case.

8.3 From PiDuce schemas to XML-Schemas, and back

The correspondence between PiDuce schemas and XML-Schema is established by suitable encoding and decoding procedures implemented by the Web interface. By encoding we mean the translation of PiDuce schemas into XML-Schema, and by decoding we mean the inverse transformation.

The precise definition of these procedures is troublesome because, although PiDuce schemas and XML-Schema have a significant common intersection, the two languages cannot be compared in terms of expressivity. There are features of XML-Schema not supported by PiDuce schemas and, conversely, features of PiDuce schemas that cannot be mapped in XML-Schema.

In this contribution, in order to ease the treatment of XML-Schemas, we focus on a subclass of them. For this reason:

- XML attributes have been ignored because they would have entangled PiDuce schemas without giving any substantial contribution to their semantic relevance;
- features such as keys, references, and facets have been ignored because they are used mainly for validation rather than for typechecking.

Encoding of PiDuce schemas into XML-Schema The encoding function $\llbracket \cdot \rrbracket$ is specified in Table 7. PiDuce schemas that have a natural representation in XML-Schema are encoded by using standard elements in the XML-Schema namespace (in Table 7, this namespace is associated with the `xsd` prefix). The remaining PiDuce schemas are encoded using extension elements in a dedicated PiDuce namespace associated with the `psd` prefix. In particular, extension elements are used for

- channel schemas, because current technologies do not provide any standard representation and description of them. We expect that this lack of expressivity will be remedied in the near future;
- schema names, because they are liberally used in PiDuce for specifying (mutually) recursive schema definitions. In XML-Schema, occurrences of symbolic names in recursive contexts must be guarded by an element;
- unions and differences of labels, because these operations have been introduced in PiDuce mostly for pattern matching rather than for typing channels. (In this case the lack of corresponding constructs in XML-Schema must not be interpreted as a weakness in XML-Schema itself.)

It is understood that any WSDL interface containing schemas with extension elements will not be compatible with standard Web services.

The encoding procedure considers every definition $U = S$ in the environment \mathcal{E} and produces a corresponding element

```
<xsd:complexType name="U">  $\llbracket S \rrbracket$  </xsd:complexType>
```

in the `wsdl:types` section of the WSDL interface.

The resulting schema may be improved for interoperability purposes by removing extension elements. We discuss two recursive patterns. The first one is *vertical recursion*, namely schema name definitions such as $U = a[U] + ()$. This definition can be encoded as

```
<xsd:complexType name="U">
  <xsd:element name="a" minOccurs="0" maxOccurs="1" type="U"/>
</xsd:complexType>
```

Another pattern is *horizontal recursion*, such as $V = a[], V + ()$ that may be encoded as

```
<xsd:complexType name="V">
  <xsd:element name="a" minOccurs="0" maxOccurs="unbound"/>
</xsd:complexType>
```

Table 7

Encoding of PiDuce schemas into XML-Schema.

<i>Standard XML-Schema</i>	
$\llbracket \perp \rrbracket$	<code><xsd:choice/></code>
$\llbracket () \rrbracket$	<code><xsd:sequence/></code>
$\llbracket a[\text{int}] \rrbracket$	<code><xsd:element name="a" type="integer"/></code>
$\llbracket a[\text{string}] \rrbracket$	<code><xsd:element name="a" type="string"/></code>
$\llbracket a[U] \rrbracket$	<code><xsd:element name="a" type="U"/></code>
$\llbracket a[S] \rrbracket$	<code><xsd:element name="a"></code> <code><xsd:complexType> $\llbracket S \rrbracket$ </xsd:complexType></code> <code></xsd:element></code>
$\llbracket S + T \rrbracket$	<code><xsd:choice> $\llbracket S \rrbracket$ $\llbracket T \rrbracket$ </xsd:choice></code>
$\llbracket L[S], T \rrbracket$	<code><xsd:sequence> $\llbracket L[S] \rrbracket$ $\llbracket T \rrbracket$ </xsd:sequence></code>
<i>XML-Schema with PiDuce extensions</i>	
$\llbracket \text{int} \rrbracket$	<code><psd:int/></code>
$\llbracket \text{string} \rrbracket$	<code><psd:string/></code>
$\llbracket n \rrbracket$	<code><psd:intlit value="n"/></code>
$\llbracket s \rrbracket$	<code><psd:stringlit value="s"/></code>
$\llbracket \langle S \rangle^\kappa \rrbracket$	<code><psd:channel capability="κ"></code> <code>$\llbracket S \rrbracket$</code> <code></psd:channel></code>
$\llbracket (a_1 + \dots + a_n)[S] \rrbracket$	<code><psd:element></code> <code><psd:with label="a_i"/> $i=1..n$</code> <code>$\llbracket S \rrbracket$</code> <code></psd:element></code>
$\llbracket (\sim \setminus (a_1 + \dots + a_n))[S] \rrbracket$	<code><psd:element></code> <code><psd:without label="a_i"/> $i=1..n$</code> <code>$\llbracket S \rrbracket$</code> <code></psd:element></code>
$\llbracket U \rrbracket$	<code><psd:schemaref name="U"/></code>

Decoding of XML-Schemas into PiDuce schemas The decoding function is defined in Table 8. The function $\llbracket X \rrbracket_S^{m,n}$ takes a fragment X of XML-Schema to be decoded, a PiDuce schema S as a continuation, and two natural numbers m and n specifying the desired repetition of the schema X , and it returns a PiDuce schema. As a side effect, the decoding function updates an environment \mathcal{E} by adding new schema name definitions (see the case $\llbracket X \rrbracket_S^{0, \text{unbound}}$ in Table 8). The two numbers m and n are determined by the attributes `minOccurs` and `maxOccurs` of XML-Schema particle elements (`maxOccurs` may

Table 8

Decoding of XML-Schema into PiDuce schemas.

$\llbracket \langle \text{xsd:element name}="a"/ \rangle \rrbracket_S$	$= a[\], S$
$\llbracket \langle \text{xsd:element name}="a" \rangle X \langle / \text{xsd:element} \rangle \rrbracket_S$	$= a[\llbracket X \rrbracket_{\text{O}}], S$
$\llbracket \langle \text{xsd:element name}="a" \text{ type}="U"/ \rangle \rrbracket_S$	$= a[U], S$
$\llbracket \langle \text{xsd:element name}="a" \text{ type}="integer"/ \rangle \rrbracket_S$	$= a[\text{int}], S$
$\llbracket \langle \text{xsd:element name}="a" \text{ type}="string"/ \rangle \rrbracket_S$	$= a[\text{string}], S$
$\llbracket \langle \text{xsd:sequence} \rangle \rrbracket_S$	$= S$
$\llbracket \langle \text{xsd:sequence} \rangle X_1 \cdots X_k \langle / \text{xsd:sequence} \rangle \rrbracket_S$	$= \llbracket X_1 \rrbracket \llbracket \langle \text{xsd:sequence} \rangle X_2 \cdots X_k \langle / \text{xsd:sequence} \rangle \rrbracket_S$
$\llbracket \langle \text{xsd:choice} \rangle \rrbracket_S$	$= \perp$
$\llbracket \langle \text{xsd:choice} \rangle X_1 \cdots X_k \langle / \text{xsd:choice} \rangle \rrbracket_S$	$= \llbracket X_1 \rrbracket_S + \cdots + \llbracket X_k \rrbracket_S$
$\llbracket \langle \text{xsd:tag minOccurs}="m" \text{ maxOccurs}="n" \rangle X_1 \cdots X_k \langle / \text{xsd:tag} \rangle \rrbracket_S$	$= \llbracket \langle \text{xsd:tag} \rangle X_1 \cdots X_k \langle / \text{xsd:tag} \rangle \rrbracket_S^{m,n}$
$\llbracket X \rrbracket_S^{0,0}$	$= S$
$\llbracket X \rrbracket_S^{0,n+1}$	$= S + \llbracket X \rrbracket_{\llbracket X \rrbracket_S^{0,n}}$
$\llbracket X \rrbracket_S^{m+1,n+1}$	$= \llbracket X \rrbracket_{\llbracket X \rrbracket_S^{m,n}}$
$\llbracket X \rrbracket_S^{0,\text{unbound}}$	$= U \quad \text{where } U \notin \text{dom}(\mathcal{E}) \text{ and } \mathcal{E} := \mathcal{E} + U = S + \llbracket X \rrbracket_U$
$\llbracket X \rrbracket_S^{m+1,\text{unbound}}$	$= \llbracket X \rrbracket_{\llbracket X \rrbracket_S^{m,\text{unbound}}}$

be **unbound** indicating unbound repetition). The case handling a particle element with these attributes set has been defined parametrically with respect to the particle's name, indicated with *tag*. Also, the function $\llbracket \cdot \rrbracket_S$ is defined on elements in the **psd** namespace, in which case it implements the inverse transformation of $\llbracket \cdot \rrbracket$. We abbreviate $\llbracket X \rrbracket_S^{1,1}$ into $\llbracket X \rrbracket_S$.

The decoding procedure considers all the complex type definitions of the form

`<xsd:complexType name="U"> X </xsd:complexType>`

that are in the section **wsdl:types** of the WSDL. For each one of such definitions the global environment \mathcal{E} is extended with

$$\mathcal{E} := \mathcal{E} + U = \llbracket X \rrbracket_{\text{O}}$$

Each time the decoding procedure is initiated, it starts with a fresh empty environment \mathcal{E} .

Remark 10 *The encoding function can be made more precise, mapping more PiDuce schemas into fully conformant XML-Schemas (for instance, $a[\mathbf{n}]$ and $a[\mathbf{s}]$ admit a conformant, albeit verbose, XML-Schema representation). On the other hand, the decoding function can be made more tolerant, recognizing more and more complex patterns in the XML-Schema representation of schemas that have a corresponding PiDuce representation (this includes the **xsd:groupref** element, the **xsd:all** element, and the derivation mechanisms).*

We notice a difference between encoding and decoding. The encoding function cannot fail: when it is applied to a PiDuce schema without a corresponding XML-Schema representation it simply resorts to the extension elements in the psd namespace. The resulting WSDL will be understandable by other PiDuce runtimes only. The decoding function, on the other hand, is partial. If the XML-Schema cannot be decoded, the decoding procedure fails and the process that initiated it is terminated.

9 The PiDuce compiler

The compiler translates a PiDuce program into an *object code* in XML format. It also type-checks the program, translates variables into indexes in the processes' environment, and removes unused variables and schema names. There are a number of advantages for having object codes in XML format. First of all, unlike binary bytecode, an XML file can be easily processed by standard XML tools (for instance DTD and XML-Schema validators). Such a file may be communicated as easily as any other XML document, even as a PiDuce value. Additionally, the XML format does not prevent a subsequent (or a just-in-time) compilation into native code and does not compromise the modularity of the whole architecture. It is also worth to remark that our choice is in accordance with XML-centered languages, such as XSLT or BPEL.

The object code closely resembles the abstract syntax tree of a PiDuce program. This is reasonable because PiDuce is meant to capture a basic set of core functionalities. We expect that a fully-featured programming language based on PiDuce would map its own constructs into this very same object code. The top-level structure of the object code produced by the compiler is the following:

```
<?xml version="1.0" encoding="utf-8"?>
<PiDuce xmlns="http://cs.unibo.it/PiDuce/opcode">
  <schemadecl>
    ...
  </schemadecl>
  <process env-size="n">
    ...
  </process>
</PiDuce>
```

Notice that the PiDuce markup is defined in the namespace `http://cs.unibo.it/PiDuce/opcode`. The `schemadecl` element contains the list of constant schema declarations used in the program. Every declaration consists of a name and an XML encoded type. For example, the declaration `Print = <File + Picture>^0` is compiled into

```

<schemadec name="Print">
  <channel capability="0">
    <choice>
      <schemaref name="File" />
      <schemaref name="Picture" />
    </choice>
  </channel>
</schemadec>

```

The compilation of schemas has already been discussed in detail in Section 8.3; in this section we focus on the **process** element. The attribute **env-size** defines the size of the environment required by the process, whereas the content of **process** is the translation of the process. For instance, the object code of the first example in Section 2 has the following process part:

```

<new index="2" name="print">
  <channel capability="IO">
    <choice>
      <schemaref name="File" />
      <schemaref name="Picture" />
    </choice>
  </channel>
  <receive service="True" index="2"
    name="print">
    <bind index="3" name="x">
      <choice>
        <schemaref name="File" />
        <schemaref name="Picture" />
      </choice>
    </bind>
    <match>
      <load index="3" name="x" />
      <branch>
        <bind index="4" name="y">
          <schemaref name="File" />
        </bind>
        <send index="1" name="printbw">
          <load index="4" name="y" />
        </send>
      </branch>
      <branch>
        <bind index="4" name="z">
          <schemaref name="Picture" />
        </bind>
        <send index="0" name="printc">
          <load index="4" name="z" />
        </send>
      </branch>
    </match>
  </service>
</new>

```

The meaning of the elements are straightforward. We only discuss the element **receive**. This element defines inputs that may be also replicated (see the attribute **service**). As any other element accessing variables in the environment, it overspecifies references by using an index – attribute **index** – and a symbolic variable – attribute **name**. While this latter attribute is not used by the virtual machine, it turns out useful for debugging and profiling object codes. The first child element of **receive** (as well as of **branch**) is a pattern that basically follows the same translation of schemas, except that it may contain the **bind** element for variable patterns. The second child element of **receive**, a **match** element in the example, is the translation of the continuation process, which is executed when a message is received.

10 Extensions

In this section we discuss a number of extensions to the **PiDuce** language that have already been analyzed and implemented or partially implemented in the current prototype.

10.1 Defining request-response services

The basic communication mechanism in the **PiDuce** programming language is asynchronous message passing. Other mechanisms, such as rendez-vous, must be explicitly programmed by means of continuation channels. This is incongruous with respect to current Web services technologies where request-response services returns results using the *same* connection. **PiDuce** partially alleviates this problem by means of an *ad-hoc* **import** instruction that allows a program to interact with a request-response Web service. This is realized by extracting the continuation channel from the sent message, and feeding it with the response received on the same connection. A similar solution can be adopted for extending **PiDuce** channel literal definitions so that the WSDL generated by the Web interface sets its interaction pattern to request-response. One possible syntax is:

new $u:S$ **reply** T **in** P

This process defines a channel literal u with a schema $\langle \mathbf{arg}[S], \langle T \rangle^0 \rangle^{I0}$, but the intention is to leave clients of this service unaware of the explicit continuation. In order to do this, when the Web interface receives a message V on a connection **socket** whose target is a local channel u with a WSDL containing a request-response pattern, the Web interface performs the following operations: (1) it verifies that the schema of V conforms with the schema of the values accepted by u ; (2) it creates a fresh channel v whose schema is adequate for communicating the response; (3) it enqueues a **ForeignRequest(socket)** in v 's request queue; (4) it invokes the method **Output(arg[V],v)** of the channel manager. When a **ForeignRequest(socket)** request is notified with a value, the value is sent on the connection identified by **socket**, the connection is closed, and v is deleted.

10.2 Channels versus services

In this contribution we always assume a one-to-one correspondence between **PiDuce** channels and WSDL resources. When defining a **PiDuce** service this assumption is seconded by the Web interface that publishes the channel as a

WSDL resource with a unique operation. This one-channel-one-service assumption falls short in faithfully modelling real Web services whose WSDL resource describes the service as a class, the operations being the methods of that class. In fact, when Web services with multiple operations are imported in a PiDuce program, only one operation is considered – the first one.

To overcome this limitation it suffices to generalize the **new** construct as follows:

$$\mathbf{new} \ s \ \{m_i : S_i\}^{i=1..n} \ \mathbf{in} \ P$$

This construct creates a service s retaining n operations. The continuation P addresses the operation m_i with $s\#m_i$. The relevant upshot of this construct is that only one WSDL resource is published and associated with the service s . Note also that s is not a PiDuce value *per se*, but each $s\#m_i$ is. The consequence of using “service” names s is that channel literals are no longer unstructured, but they consist of two parts, one addressing the location of the corresponding WSDL resource, and one specifying the operation.

Symmetrically, the **import** construct may also be extended:

$$\mathbf{import} \ s \ \{m_i : S_i\}^{i=1..n} = \mathbf{v} \ \mathbf{in} \ P$$

Here \mathbf{v} is the address of the WSDL resource corresponding to the service being imported. This instruction is successful provided that the WSDL resource at \mathbf{v} contains *at least* the operations m_i , and if the schema constraints are satisfied as described in Section 6. As before, in P , each operation m_i is addressed by $s\#m_i$.

10.3 Join patterns

In concurrent programming and Web services it is often necessary to orchestrate two or more parties. The **select** operation in PiDuce already provides a basic form of orchestration by permitting the execution of exactly one party out of several ones. However, it falls short in modelling orchestration patterns where some parties require the simultaneous availability of messages from different channels, or if their execution depends upon the number of messages available on the same channel. In this section we analyze a thoroughly studied extension, the so-called *join-pattern* [17], which permits a natural modelling of synchronization and competition patterns. The formal development of join-patterns in the context of PiDuce-like machines has been carried out in [23].

A join-pattern links several inputs to one continuation, the idea being that the continuation is activated only when all the inputs are satisfied. More complex

orchestrations are usually built by selecting a join pattern out of a set of them. Join-patterns may be added to PiDuce by extending the syntax with the following rules:

$$\begin{aligned} P &::= \dots \mid \text{join } \{J_i \triangleright P_i^{i \in 1..n}\} \\ J &::= u?(F) \mid J \& J \end{aligned}$$

For example the process

```
bw_printer_twice?(a[f : Pdf], b[done : ⟨()⟩0, c[abort : ⟨()⟩0]).
  new done1 : ⟨()⟩0, done2 : ⟨()⟩0, abort1 : ⟨()⟩0, abort2 : ⟨()⟩0 in
    spawn {bw_printer1!(a[f], b[done1], c[abort1])}
    spawn {bw_printer2!(a[f], b[done2], c[abort2])}
    join { done1?(()) & done2?(()) ▷ done!()
          abort1?(()) ▷ abort!()
          abort2?(()) ▷ abort!() }
```

defines a `bw_printer_twice` service that accepts requests for printing a Pdf document twice and it forwards the request in parallel to two different printers. The client is notified on the channel `done` only when *both* printers have finished their job, whereas it is notified on the channel `abort` as soon as *one* of the printers fails.

Join patterns have a deep impact on the implementation of the PiDuce channel manager: a message delivered to a channel with a nonempty request queue is not necessarily consumed, because the corresponding inputs may depend upon the state of other channels if they occur in join patterns. This means that a channel may now have at the same time nonempty message and request queues. More importantly, determining that a continuation is to be triggered entails examining the message queues of all the channels involved in a join pattern. This operation is plausible as long as all the channels in a join pattern are owned by the same channel manager, that is if they are all *co-located*. In the join calculus this is no issue, since channel creation and join patterns are represented by the same linguistic construct. Apart from this aspect, our implementation of join patterns is close to the one in join calculus where they are compiled into a finite state automata recording the state of the message queues of the subjects [15]. Readers are referred to [23] for the details.

11 Related work

The PiDuce prototype falls within the domain of distributed abstract machines for pi-like calculi. Among them we recall Facile [20], the Jocaml prototype [13], Distributed pi calculus [4], Nomadic Pict [36], the Ambient Calculus [9,35]. The differences between our model and the other ones are as follows. Facile

uses two classes of distributed entities: (co-)located processes which execute, and channel-managers which mediate interaction. This forces it to use a handshake discipline for communication. Jocasml simplifies the model by combining input processes with channel-managers. However, it uses a quite different form of interaction, which does not relate that closely to pi calculus communication. It also forces a coarser granularity, in which every channel must be co-located with at least one other. Unlike Jocasml, our machine has finer granularity and uses the same form of interaction as the π -calculus. The other models add explicit location constructs to the π -calculus and use agent migrations for remote interactions.

PiDuce has been strongly influenced by XDuce, a functional language for XML processing [21]. In XDuce, values do not carry channels, and the subschema relation is never needed at run-time. Our paper may also be read as an investigation of the extension of XDuce values and schemas with channels.

Several integrations of processes and semi-structured data have been studied in recent years. Two similar contributions, that are contemporary and independent to this one, are [11,7]. The schema language in [11] is the one of [5] plus the channel constructors for input, output, and input-output capability. No apparent restriction to reduce the computational complexity of pattern matching is proposed. The schema language of [2] is simpler than that of PiDuce. In particular recursion is omitted and labeled schemas have singleton labels.

Other contributions integrating semi-structured data and processes are discussed in order. TulaFale [6], a process language with XML data, is especially designed to address web services security issues such as vulnerability to XML rewriting attacks. The language has no static semantics. The integration of PiDuce with the security features of TulaFale seems a promising direction of research. Xd π [19] is a language that supports dynamic web page programming. This language is π -calculus with locations plus the explicit primitives for process migration, for updating data, and for running a script. The emphasis of Xd π is towards behavioral equivalences and analysis techniques for behavioral properties. A similar contribution to [19] is Active XML [1] that uses an underlying model consisting of a set of peer locations with data and services.

12 Conclusions

In this contribution we have presented the PiDuce project, a distributed implementation of the asynchronous π -calculus with tree-structured datatypes and pattern matching. The resulting language is suitable for programming Web services, and this motivates our choice of XML idioms, such as XML-Schema and

WSDL for types and interfaces, respectively. In this respect, PiDuce fills the gap between theory and practice by formally defining a programming language and showing its implementation using industrial standards.

Regarding the description of Web services interfaces, it is remarkable that WSDL 1.1 (already published as a W3C Note [29]) does not consider service references as first class values, that is natural in a distributed setting, in π -calculus, and, thereafter, in PiDuce. This lack of expressivity has been at least partly amended in WSDL 2.0 [30,31] that, at the time of this writing, is in a Working Draft status. Still, we note significant differences between our approach and the way “Web services as values” are handled in WSDL 2.0. For example the client receiving a service reference u must eventually compare the schema in the WSDL of u with some local schema before using it or forward u to a third party. While this comparison, called subschema relation in this paper, is fundamental in PiDuce, it has been completely overlooked in WSDL 2.0. Another difference is in the meaning of service references. In Section 10.2 we propose to identify service references, called channel literals in this paper, with the location of the Web service *and* the operation to be invoked. Doing so allows us to reuse PiDuce type-system by simply adding structure to channel literals. In WSDL 2.0, on the other hand, it is the whole Web service, not a specific operation, that is communicated, and it is up to the receiver to invoke the desired operation(s). Modeling this behavior would have required substantial changes to our formal system. While we are not advocating *our* point of views as the correct ones, we believe that projects like PiDuce may provide committees with useful insights and ideas for the standardization of technologies in a more disciplined way.

Few remarks about XML-Schema are in order. First of all there is a large overlapping between XML-Schema and PiDuce schemas, which has been formalized in Section 8. Apart from channel schemas, the other major departure from XML schema is the support for nondeterministic labelled schemas. These schemas make the computational complexity of the subschema relation exponential, but they are essential for the static semantics of a basic operator in PiDuce, the pattern-matching (see the third premise of rule (MATCH) in Table 3). Noticeably, the constraint of labelled-determinedness on channel schemas guarantees a polynomial cost for the subschema relation (and for the pattern matching) at runtime.

Future work in the PiDuce project is planned in two directions: the first direction is rather pragmatic, and is aimed to improving interoperability and support to existing protocols. The goal is to interface PiDuce with some real-world Web services and to carry on some practically useful experimentation. The other direction regards conceptual features that are desirable and that cannot be expressed conveniently in the current model. In particular error handling and transactional mechanisms. These mechanisms, which are basic

in BPEL, permit the coordination of processes located on different machines by means of time constraints. This is a well-known problematic issue in concurrency theory. An initial investigation about transactions in the setting of the asynchronous π -calculus has been undertaken in [24]. A core BPEL language without such advanced coordination mechanisms should be compilable in PiDuce without much effort, thus equipping BPEL with a powerful static semantics. We expect to define a translation in the next future.

Another direction of research is about dynamic XML data, namely those data containing active parts that may be executed on clients' machines. This is obtained by transmitting processes during communications, a feature called process migration. The PiDuce prototype disallows program deployments on the network. However, the step towards migration is quite short due to the fact that object code is in XML format. Therefore it suffices to introduce two new schemas: the object code schema and the environment schema, and admit channels carrying messages of such schemas.

References

- [1] S. Abiteboul, O. Benjelloun, I. Manolescu, T. Milo, and R. Weber. Active XML: Peer-to-peer data and Web services integration. In *Proceedings of the Twenty-Eighth International Conference on Very Large Data Bases (VLDB 2002)*, Hong Kong SAR, China, pages 1087–1090. Morgan Kaufmann Publishers, 2002.
- [2] L. Acciai and M. Boreale. XPi: a typed process calculus for XML messaging. In *7th Formal Methods for Object-Based Distributed Systems (FMOODS'05)*, volume 3535 of *Lecture Notes in Computer Science*, pages 47 – 66. Springer-Verlag, 2005.
- [3] R. Amadio. An asynchronous model of locality, failure, and process mobility. In D. Garlan and D. L. Métayer, editors, *Proceedings of COORDINATION 1997*, volume 1282 of *Lecture Notes in Computer Science*, pages 374–391. Springer-Verlag, 1997.
- [4] R. Amadio, G. Boudol, and C. Lhoussaine. The receptive distributed pi-calculus (extended abstract). In C. Pandu Rangan, V. Raman, and R. Ramanujam, editors, *Proceedings of Foundations of Software Technology and Theoretical Computer Science (FSTTCS 1999)*, volume 1738 of *Lecture Notes in Computer Science*, pages 304–315. Springer-Verlag, 1999.
- [5] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. In *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming (ICFP-03)*, pages 51–63, New York, 2003. ACM Press.
- [6] K. Bhargavan, C. Fournet, A. Gordon, and R. Pucella. TulaFale: A Security Tool for Web Services. In *Proceedings of the 2nd International Symposium*

on *Formal Methods for Components and Objects (FMCS'03)*, volume 3188 of *LNCS*, pages 197–222. Springer-Verlag, 2004.

- [7] M. Boreale. On the expressiveness of internal mobility in name-passing calculi. *Theor. Comput. Sci.*, 195(2):205–226, 1998.
- [8] G. Boudol. Asynchrony and the π -calculus (note). Rapport de Recherche 1702, INRIA Sophia-Antipolis, 1992.
- [9] L. Cardelli and A. D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000.
- [10] S. Carpineti and C. Laneve. A basic contract language for Web services. In *Proceedings of the European Symposium on Programming (ESOP 2006)*, LNCS. Springer-Verlag, 2006. (to appear).
- [11] G. Castagna, R. D. Nicola, and D. Varacca. Semantic subtyping for the π -calculus. In *20th IEEE Symposium on Logic in Computer Science (LICS'05)*. IEEE Computer Society, 2005.
- [12] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available at <http://www.grappa.univ-lille3.fr/tata>, 1997. released October, 1st 2002.
- [13] S. Conchon and F. L. Fessant. Jocaml: Mobile agents for objective-caml. In *First International Symposium on Agent Systems and Applications Third International Symposium on Mobile Agents*, pages 22–29. IEEE Computer Society Press, October, 1999.
- [14] F. Curbera, Y. Golland, J. Klein, F. Leyman, D. Roller, S. Thatte, and S. Weerawarana. Business process execution language for web services (bpel4ws 1.0). Available at <http://www.106.ibm.com/developerworks/webservices/library/ws-bpel/>, 2002.
- [15] F. L. Fessant and L. Maranget. Compiling join-patterns. In U. Nestmann and B. C. Pierce, editors, *HLCL '98: High-Level Concurrent Languages*, volume 16(3) of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers., 1998.
- [16] C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of POPL 1996*, pages 372–385. ACM, ACM Press, 1996.
- [17] C. Fournet and G. Gonthier. A hierarchy of equivalences for asynchronous calculi. In *Proceedings of 25th Colloquium on Automata, Languages and Programming (ICALP)*, volume 1443 of *LNCS*, pages 844–855. Springer-Verlag, 1998.
- [18] P. Gardner, C. Laneve, and L. Wischik. Linear forwarders. In *14th International Conference on Concurrency Theory (CONCUR 2003)*, volume 2761 of *Lecture Notes in Computer Science*, pages 415 – 430. Springer-Verlag, 2002.

- [19] P. Gardner and S. Maffei. Modelling dynamic web data. *Theoretical Computer Science*, 342:104–131, 2005.
- [20] A. Giacalone, P. Mishra, and S. Prasad. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, 1989.
- [21] H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology (TOIT)*, 3(2):117–148, 2003.
- [22] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. *ACM SIGPLAN Notices*, 35(9):11–22, 2000.
- [23] C. Laneve and L. Padovani. Smooth orchestrators. In *Proceedings of Foundations of Software Science and Computation Structures (FOSSACS 2006)*, LNCS. Springer-Verlag, 2006. (to appear).
- [24] C. Laneve and G. Zavattaro. Foundations of Web Transactions. In *Proceedings of Foundations of Software Science and Computation Structures (FOSSACS’05)*, volume 3441 of *LNCS*, pages 282–298. Springer-Verlag, 2005.
- [25] Microsoft Corporation. Biztalk server. <http://www.microsoft.com/biztalk/>.
- [26] W3C XML Schema Working Group. XML Schema Part 0: Primer Second Edition. Available at <http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/>. W3C Recommendation - October, 28th 2004.
- [27] W3C XML Schema Working Group. XML Schema Part 1: Structures Second Edition. Available at <http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/structures.html>. W3C Recommendation - October, 28th 2004.
- [28] W3C XML Schema Working Group. XML Schema Part 2: Datatypes Second Edition. Available at <http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html>. W3C Recommendation - October, 28th 2004.
- [29] Web Services Description Working Group. Web Services Description Language (WSDL) 1.1). Available at <http://www.w3.org/TR/2001/NOTE-wsdl-20010315>. W3C Note 15 March 2001.
- [30] Web Services Description Working Group. Web Services Description Language (WSDL) Version 2.0 Part 0: Primer. Available at <http://www.w3.org/TR/2005/WD-wsdl20-primer-20050803/>. W3C Working Draft 3 August 2005.
- [31] Web Services Description Working Group. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. Available at <http://www.w3.org/TR/2005/WD-wsdl20-20050803/>. W3C Working Draft 3 August 2005.
- [32] XML Core Working Group. Extensible Markup Language (XML) 1.1. Available at <http://www.w3.org/TR/2004/REC-xml11-20040204/>. 04 February 2004, edited in place 15 April 2004.

- [33] M. Merro and D. Sangiorgi. On asynchrony in name-passing calculi. In K. G. Larsen, S. Skyum, and G. Winskel, editors, *Proceedings of 25th International Colloquium on Automata, Languages, and Programming (ICALP 1998)*, volume 1443 of *Lecture Notes in Computer Science*, pages 856–867. Springer-Verlag, 1998.
- [34] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part I/II. *Journal of Information and Computation*, 100:1–77, Sept. 1992.
- [35] A. Phillips, N. Yoshida, and S. Eisenbach. A distributed abstract machine for boxed ambient calculi. In *Proceedings of the European Symposium on Programming (ESOP 2004)*, LNCS, pages 155–170. Springer-Verlag, Apr. 2004.
- [36] P. Sewell, P. Wojciechowski, and B. Pierce. Location independence for mobile agents. In H. E. Bal, B. Belkhouche, and L. Cardelli, editors, *ICCL 1998*, volume 1686 of *Lecture Notes in Computer Science*, pages 1–31. Springer-Verlag, 1999.

A Soundness of the static semantics

In the following demonstrations we use three properties of \lesssim : (a) \lesssim is contravariant with respect to $\langle \cdot \rangle^0$, (a) \lesssim is covariant with respect to $\langle \cdot \rangle^I$, (a) \lesssim is invariant with respect to $\langle \cdot \rangle^{I0}$. These properties are immediate consequences of the definition.

The basic statements below are standard preliminary results for the subject reduction theorem.

Lemma 11 (*Weakening*)

- (1) If $\Gamma \vdash E : S$ and $u \notin \text{fn}(E)$ then $\Gamma + u : T \vdash E : S$;
- (2) If $\Gamma ; \Delta \vdash P$ and $u \notin \text{fn}(P)$ then (a) $(\Gamma ; \Delta) + u : S \vdash P$, (b) $\Gamma + u : \langle S \rangle^\kappa ; \Delta + u : \langle S \rangle^{I0} \vdash P$;
- (3) If $\Gamma + u : \langle S \rangle^\kappa ; \Delta \vdash P$ and $u \notin \text{dom}(\Delta)$ then $\Gamma + u : \langle S \rangle^\kappa ; \Delta + u : \langle S \rangle^{I0} \vdash P$.

The last statement of Lemma 11 is peculiar of PiDuce typing system. It states that well-typing is preserved if channel literals and variables with channel schemas are made local (i.e. when the capability is changed into I0).

Lemma 12 (*Strengthening*) If $\Gamma + u : S \vdash E : T$ and $u \notin \text{fn}(E)$ then $\Gamma \vdash E : T$. Similarly for $\Gamma ; \Delta \vdash P$.

Lemma 13 (*Substitution*)

- (1) Let $\Gamma \vdash E : T$ and $\Gamma \vdash u : S$. If $\Gamma \vdash V : S'$ and $S' \lesssim S$ then $\Gamma \vdash E\{V/u\} : T'$ with $T' \lesssim T$.

(2) Let $\Gamma; \Delta \vdash P$ and $\Gamma + \Delta \vdash u : S$. If $\Gamma \vdash V : T$, and $T \lesssim S$ then $\Gamma; \Delta \vdash P\{V/u\}$.

Proof. The demonstration is by induction on the structures of the proofs of $\Gamma \vdash E : T$ and $\Gamma; \Delta \vdash P$.

For (1) we only discuss the case when E is a sequence and $\Gamma \vdash a[E'], E''$. By the hypothesis $\Gamma \vdash E' : T''$ and $\Gamma \vdash E'' : T'''$, and by inductive hypothesis we have

$$\Gamma \vdash E'\{V'/u\} : R \quad R \lesssim T'' \quad (\text{A.1})$$

$$\Gamma \vdash E''\{V'/u\} : R' \quad R' \lesssim T''' \quad (\text{A.2})$$

From (A.1), and (A.2) we obtain $T' = a[R], R'$. We conclude with (LSEQ).

For (2) we only discuss the case when the last rule is (OUT). Then $P = w!(E)$ and the premises of the rule are the judgments $\Gamma \vdash E : S'$ and $\Gamma; \Delta \vdash w : R$, and the predicate

$$R \lesssim \langle S' \rangle^0 \quad (\text{A.3})$$

We must prove $\Gamma; \Delta \vdash w!(E)\{V/u\}$. By $\Gamma \vdash E : S'$, the hypothesis $\Gamma \vdash V : T$, $T \lesssim S$, and the substitution lemma for values, we obtain

$$\Gamma \vdash E\{V/u\} : T' \quad (\text{A.4})$$

$$T' \lesssim S' \quad (\text{A.5})$$

As regards the subject of the output, there are two subcases: (a) $u \neq w$ and (b) $u = w$. Case (a) follows by (A.3), (A.5), contravariance of $\langle \cdot \rangle^0$ and transitivity of \lesssim . Case (b) implies $S = R$ and follows by (A.4), by the hypotheses $\Gamma \vdash u : S$, by (A.3), the contravariance of $\langle \cdot \rangle^0$, and the transitivity of \lesssim . \square

The soundness of pattern matching is established by the next lemma.

Lemma 14 (*Pattern Matching*)

- (1) If $\Gamma \vdash V \in F \rightsquigarrow \sigma$ and $u \notin \text{fn}(V)$ then $\Gamma + u : S \vdash V \in F \rightsquigarrow \sigma$;
- (2) If $\Gamma \vdash V : S$ and $\Gamma \vdash V \in F \rightsquigarrow \sigma$ then, for every $u \in \text{fn}(F)$, $\Gamma \vdash \sigma(u) : T$ and $T \lesssim \text{Env}(F)(u)$;

The preliminaries are in place for the subject reduction theorem. For readability sake we also recall the statement.

Theorem 7 (*Subject Reduction*) Let $\Gamma; [\Gamma]_1^{\text{I0}} \vdash P$. Then

- (1) if $\Gamma \vdash_1 P \xrightarrow{(\Gamma')u!(V)} Q$ then (a) $\Gamma + \Gamma'; [\Gamma + \Gamma']_1^{I0} \vdash Q$, (b) $\Gamma + [\Gamma]_1^{I0} \vdash u:S$, $\Gamma + \Gamma' \vdash V:T$ and $S \lesssim \langle T \rangle^0$;
- (2) if $\Gamma \vdash_1 P \xrightarrow{u?(F)} Q$ then (a) $(\Gamma; [\Gamma]_1^{I0}) + \text{Env}(F) \vdash Q$ and (b) $\Gamma + [\Gamma]_1^{I0} \vdash u:S$ with $S \lesssim \langle \text{schof}(F) \rangle^I$;
- (3) if $\Gamma \vdash_1 P \xrightarrow{(\Gamma')u \circ v} Q$ then (a) $\Gamma + \Gamma'; [\Gamma + \Gamma']_1^{I0} \vdash Q$ and (b) $\Gamma \setminus \text{dom}([\Gamma + \Gamma']_1^{I0}) \vdash u:S$, $\Gamma + \Gamma' \vdash v:\langle T \rangle^0$ and $S \lesssim \langle T \rangle^I$;
- (4) if $\Gamma \vdash_1 P \xrightarrow{(u@1':\langle S \rangle^{I0})} Q$ then $(\Gamma; [\Gamma]_1^{I0}) + u:\langle S \rangle^{I0} \vdash Q$;
- (5) if $\Gamma \vdash_1 P \xrightarrow{v:S} Q$ then (a) $\Gamma \vdash v:\langle T \rangle^\kappa$ and (b) $(\Gamma; [\Gamma]_1^{I0}) + v:\langle T \rangle^\kappa \vdash Q$ with $\langle T \rangle^\kappa \lesssim S$;
- (6) if $\Gamma \vdash_1 P \xrightarrow{\tau} Q$ then $\Gamma; [\Gamma]_1^{I0} \vdash Q$.

Let $\vdash \mathbf{M}$ if, for every $\Gamma \vdash_1 P$ in \mathbf{M} : $\Gamma; [\Gamma]_1^{I0} \vdash P$. Then

- (7) if $\vdash \mathbf{M}$ and $\mathbf{M} \longrightarrow \mathbf{N}$ then $\vdash \mathbf{N}$.

Proof. The demonstration proceeds by induction on the structure of the proof of $\Gamma \vdash_1 P \xrightarrow{\mu} Q$ and by cases on the last rule that has been applied for the first six items. (7) is similar, but the induction is on the structure of the proof $\vdash \mathbf{M}$. We omit the cases that are straightforward.

When the last rule is an instance of (TR4) we have:

$$\frac{\Gamma + v:\langle S \rangle^\kappa \vdash_1 P \xrightarrow{(\Gamma')u!(V)} Q \quad v \neq u \quad v \in \text{fn}(V) \setminus \text{dom}(\Gamma')}{\Gamma \vdash_1 \text{new } v:\langle S \rangle^\kappa \text{ in } P \xrightarrow{(\Gamma'+v:\langle S \rangle^\kappa)u!(V)} Q}$$

By inductive hypotheses applied to $\Gamma + v:\langle S \rangle^\kappa \vdash_1 P \xrightarrow{(\Gamma')u!(V)} Q$ we obtain

$$\Gamma + v:\langle S \rangle^\kappa + \Gamma'; [\Gamma + v:\langle S \rangle^\kappa + \Gamma']_1^{I0} \vdash Q \tag{A.6}$$

$$\Gamma + v:\langle S \rangle^\kappa + [\Gamma + v:\langle S \rangle^\kappa]_1^{I0} \vdash u:S' \tag{A.7}$$

$$\Gamma + v:\langle S \rangle^\kappa + \Gamma' \vdash V:T \tag{A.8}$$

$$S' \lesssim \langle T \rangle^0 \tag{A.9}$$

The conclusion (a) follows directly from (A.6); the conclusion (b) follows by (A.7), (A.8), and (A.9) because $u \neq v$.

When the last rule is an instance of (TR8) we have:

$$\frac{\Gamma \vdash_1 P \xrightarrow{(\Gamma')u!(V)} P' \quad \Gamma \vdash_1 Q \xrightarrow{u?(F)} Q' \quad \text{dom}(\Gamma') \cap \text{fn}(Q) = \emptyset \quad \Gamma + \Gamma' \vdash V \in F \rightsquigarrow \sigma}{\Gamma \vdash_1 \text{spawn } \{P\} Q \xrightarrow{\tau} \text{new } \Gamma' \text{ in spawn } \{P'\} Q' \sigma}$$

By inductive hypotheses on $\Gamma \vdash P \xrightarrow{(\Gamma')u!(V)} P'$ and $\Gamma \vdash Q \xrightarrow{u?(F)} Q'$ we have:

$$\Gamma + \Gamma'; [\Gamma + \Gamma']_1^{I0} \vdash P' \quad (\text{A.10})$$

$$\Gamma + \Gamma' \vdash V : T \quad (\text{A.11})$$

$$\Gamma + [\Gamma]_1^{I0} \vdash u : S \quad S \lesssim \langle \text{schof}(F) \rangle^I \quad S \lesssim \langle T \rangle^0 \quad (\text{A.12})$$

$$(\Gamma; [\Gamma]_1^{I0}) + \text{Env}(F) \vdash Q' \quad (\text{A.13})$$

By Lemma 14(1,3) applied to $\Gamma + \Gamma' \vdash V \in F \rightsquigarrow \sigma$, (A.11), and (A.12) we obtain that, for every $v \in \text{fn}(F)$, $\Gamma + \Gamma' \vdash \sigma(v) : T'$ and $T' \lesssim \text{Env}(F)(v)$. By Lemma 13 applied to this last judgment, (A.12), and (A.13) we derive $(\Gamma; [\Gamma]_1^{I0}) + \Gamma' \vdash Q'\sigma$. We conclude with (A.10), (SPAWN), and (NEW).

The case (TR10) is omitted because the resulting process is complex and the demonstration requires a long uninteresting analysis of the proof tree.

When the last rule is an instance of (DTR1) we have:

$$\begin{array}{c} \text{(DTR1)} \\ \hline \Gamma \vdash_1 P \xrightarrow{(v_i S_i^{i \in I})u!(V)} Q \quad u@1' \quad (\mathbf{w}_i @1 \quad \mathbf{w}_i \notin \text{dom}(\Gamma))^{i \in I} \quad \Gamma'' = \Gamma|_{\text{fn}(V) \setminus \{v_i^{i \in I}\}} \\ \hline \Gamma \vdash_1 P \parallel \Gamma' \vdash_1 R \longrightarrow \\ \Gamma + \mathbf{w}_i : S_i^{i \in I} \vdash_1 Q\{\mathbf{w}_i / v_i^{i \in I}\} \parallel \Gamma' + \mathbf{w}_i : S_i^{i \in I} + \Gamma'' \vdash_1 \text{spawn } \{u!(V\{\mathbf{w}_i / v_i^{i \in I}\})\} R \end{array}$$

by inductive hypothesis on $\Gamma \vdash_1 P \xrightarrow{(v_i S_i^{i \in I})u!(V)} Q$ we obtain

$$\Gamma + v_i : S_i^{i \in I}; [\Gamma + v_i : S_i^{i \in I}]_1^{I0} \vdash Q \quad (\text{A.14})$$

$$\Gamma + [\Gamma]_1^{I0} \vdash u : S \quad (\text{A.15})$$

$$\Gamma + v_i : S_i^{i \in I} \vdash V : T \quad S \lesssim \langle T \rangle^0 \quad (\text{A.16})$$

The conclusion $\Gamma + \mathbf{w}_i : S_i^{i \in I}; [\Gamma + \mathbf{w}_i : S_i^{i \in I}]_1^{I0} \vdash Q\{\mathbf{w}_i / v_i^{i \in I}\}$ follows by (A.14), Lemma 13, and the premise $\mathbf{w}_i @1$. As regard the judgment

$$\Gamma' + \mathbf{w}_i : S_i^{i \in I} + \Gamma''; [\Gamma' + \mathbf{w}_i : S_i^{i \in I} + \Gamma'']_1^{I0} \vdash \text{spawn } \{u!(V\{\mathbf{w}_i / v_i^{i \in I}\})\} R \quad (\text{A.17})$$

by $\vdash \mathbf{M}$ we derive $\Gamma'; [\Gamma']_1^{I0} \vdash R$. Then by Lemma 11 and the localization property of \mathbf{M} we obtain $\Gamma' + \mathbf{w}_i : S_i^{i \in I} + \Gamma''; [\Gamma' + \mathbf{w}_i : S_i^{i \in I} + \Gamma'']_1^{I0} \vdash R$. To demonstrate

$$\Gamma' + \mathbf{w}_i : S_i^{i \in I} + \Gamma''; [\Gamma' + \mathbf{w}_i : S_i^{i \in I} + \Gamma'']_1^{I0} \vdash u!(V\{\mathbf{w}_i / v_i^{i \in I}\}) \quad (\text{A.18})$$

we reason as follows (the theorem follows by (SPAWN) applied to (A.17) and (A.18)). By (A.15), $u@1'$, and the consistency of Γ' we derive

$$\Gamma' + \mathbf{w}_i : S_i^{i \in I} + \Gamma'' \vdash u : S \quad (\text{A.19})$$

By (A.16) and Lemma 12 we derive $v_i : S_i^{i \in I} + \Gamma'' \vdash V : T$. By this judgment it is easy to derive $\Gamma' + v_i : S_i^{i \in I} + \Gamma'' \vdash V : T$. By Lemmas 11, 13, and 12 we obtain

$$\Gamma' + \mathbf{w}_i : S_i^{i \in I} + \Gamma'' \vdash V\{\mathbf{w}_i / v_i^{i \in I}\} : T' \quad (\text{A.20})$$

with $T' \lesssim T$. By $T' \lesssim T$, the contravariance of $\langle \cdot \rangle^I$, the transitivity of \lesssim , and $S \lesssim \langle T \rangle^0$ we yield $S \lesssim \langle T' \rangle^0$. This last relation, (A.19), and (A.20) allow us to apply (OUT) and conclude.

When the last rule is an instance of (DTR2) we have:

$$\begin{array}{c} (\text{DTR2}) \\ \hline \Gamma \vdash_1 P \xrightarrow{(u@1':\langle S \rangle^{I0})} Q \quad \mathbf{v}@1' \quad \mathbf{v} \notin \text{dom}(\Gamma') \\ \hline \Gamma \vdash_1 P \parallel \Gamma' \vdash_{1'} R \longrightarrow \Gamma + \mathbf{v} : \langle S \rangle^{I0} \vdash_1 Q\{\mathbf{v}/u\} \parallel \Gamma' + \mathbf{v} : \langle S \rangle^{I0} \vdash_{1'} R \end{array}$$

We prove $\Gamma + \mathbf{v} : \langle S \rangle^{I0} \vdash_1 Q\{\mathbf{v}/u\}$. By the inductive hypothesis on $\Gamma \vdash_1 P \xrightarrow{(u@1':\langle S \rangle^{I0})} Q$ we obtain $\Gamma; [\Gamma_1]_{I0} + u : \langle S \rangle^{I0} \vdash Q$. By Lemma 11(3) and Lemma 11(2) we obtain $\Gamma + u : \langle S \rangle^{I0} + \mathbf{v} : \langle S \rangle^{I0}; [\Gamma]_1^{I0} + u : \langle S \rangle^{I0} + v : \langle S \rangle^{I0} \vdash Q$. From this and Lemma 13 we derive $\Gamma + u : \langle S \rangle^{I0} + \mathbf{v} : \langle S \rangle^{I0}; [\Gamma]_1^{I0} + u : \langle S \rangle^{I0} + v : \langle S \rangle^{I0} \vdash Q\{\mathbf{v}/u\}$. By Lemma 12 and $\mathbf{v}@1'$ we conclude $\Gamma + \mathbf{v} : \langle S \rangle^{I0}; [\Gamma]_1^{I0} \vdash Q\{\mathbf{v}/u\}$. We prove $\Gamma' + \mathbf{v} : \langle S \rangle^{I0} \vdash_{1'} R$. By the hypothesis $\Gamma'; [\Gamma']_{1'}^{I0} \vdash R$, $\mathbf{v} \notin \text{dom}(\Gamma')$, $\mathbf{v} \notin \text{fn}(R)$ we conclude $\Gamma' + v : \langle S \rangle^{I0}; [\Gamma' + v : \langle S \rangle^{I0}]_{1'}^{I0} \vdash R$ by Lemma 11(2).

When the last rule is an instance of (DTR3) we have

$$\begin{array}{c} (\text{DTR3}) \\ \hline \Gamma \vdash_1 P \xrightarrow{u:S} Q \quad \mathbf{u}@1' \quad \Gamma' \vdash \mathbf{u} : \langle S' \rangle^\kappa \quad \langle S' \rangle^\kappa \lesssim S \\ \hline \Gamma \vdash_1 P \parallel \Gamma' \vdash_{1'} R \longrightarrow \Gamma + \mathbf{u} : \langle S' \rangle^\kappa \vdash_1 Q \parallel \Gamma' \vdash_{1'} R \end{array}$$

$\Gamma + \mathbf{u} : \langle S' \rangle^\kappa; [\Gamma]_1^{I0} \vdash Q$ follows by the inductive hypothesis on $\Gamma \vdash_1 P \xrightarrow{u:S} Q$ and $\mathbf{u}@1'$.

When the last rule is an instance of (DTR4) we have

$$\begin{array}{c} (\text{DTR4}) \\ \hline \Gamma \vdash_1 P \xrightarrow{(\Gamma'')^{\mathbf{u} \circ v}} Q \quad \mathbf{u}@1' \quad \Gamma' \vdash \mathbf{u} : \langle S \rangle^\kappa \quad \text{dom}(\Gamma'') \cap \text{dom}(\Gamma') = \emptyset \quad \Gamma''' = \Gamma|_{\{v\}} + \Gamma'' \\ \hline \Gamma \vdash_1 P \parallel \Gamma' \vdash_{1'} R \longrightarrow \Gamma + \Gamma'' \vdash_1 Q \parallel \Gamma' + \Gamma''' \vdash_{1'} \text{spawn } \{\mathbf{u}?(x : S) \ v!(x)\} R \end{array}$$

by inductive hypothesis on $\Gamma \vdash_1 P \xrightarrow{(\Gamma'')^{\mathbf{u} \circ v}} Q$ we obtain

$$\Gamma + \Gamma''; [\Gamma + \Gamma'']_1^{I_0} \vdash Q \quad (\text{A.21})$$

$$\Gamma \setminus \text{dom}([\Gamma + \Gamma'']_1^{I_0}) \vdash u : T \quad (\text{A.22})$$

$$\Gamma + \Gamma'' \vdash v : \langle T' \rangle^0 \quad (\text{A.23})$$

$$T \lesssim \langle T' \rangle^I \quad (\text{A.24})$$

By (A.21) we immediately derive that the leftmost machine is well-typed. Therefore we focus on the rightmost machine. To demonstrate the correctness of its process we will eventually use (SPAWN). Therefore we are reduced to prove: (1) $\Gamma' + \Gamma'''; [\Gamma' + \Gamma''']_1^{I_0} \vdash R$ and (2) $\Gamma' + \Gamma'''; [\Gamma' + \Gamma''']_1^{I_0} \vdash u?(x : S)v!(x)$. If $v \notin \text{fn}(R)$, the judgment (1) follows by the hypothesis $\text{dom}(\Gamma'') \cap \text{dom}(\Gamma') = \emptyset$, by $\Gamma'; [\Gamma']_{I_0} \vdash R$, and by Lemma 11. If $v \in \text{fn}(R)$, (1) follows by consistency of Γ' and Lemma 11. As regard the judgment (2), since Γ is consistent, $T = \langle S \rangle^\kappa$. Since $v \in \text{dom}(\Gamma''')$, by (A.23) it is easy to derive the judgement $\Gamma' + \Gamma''' + x : S \vdash v : \langle T' \rangle^0$. By (A.24), $S \lesssim T'$, contravariance of $\langle \cdot \rangle^0$, we have $\langle T' \rangle^0 \lesssim \langle S \rangle^0$. Then, by the rule (OUT), $\Gamma' + \Gamma''' + x : S; [\Gamma' + \Gamma''' + x : S]_1^{I_0} \vdash v!(x)$. Finally, from the hypotheses $\Gamma' \vdash u : \langle S \rangle^\kappa$, (A.24), and $\text{dom}(\Gamma'') \cap \text{dom}(\Gamma') = \emptyset$ we derive $\Gamma' + \Gamma''' + [\Gamma' + \Gamma''']_1^{I_0} \vdash u : \langle S \rangle^\kappa$ with $\langle S \rangle^\kappa \lesssim \langle S \rangle^I$. We conclude with (SELECT). \square

The demonstration of the Progress Theorem follows.

Theorem 9 (Progress) *Let Γ be channeled and let $\Gamma; [\Gamma]_1^{I_0} \vdash P$. If $\Gamma \vdash_1 P \xrightarrow{(\Gamma')u!(V)} Q'$ and $\Gamma \vdash_1 P \xrightarrow{u?(F)} Q''$ then there is Q such that $\Gamma \vdash_1 P \xrightarrow{\tau} Q$.*

Proof: We begin by demonstrating that there is σ such that $\Gamma \vdash V \in F \rightsquigarrow \sigma$. By Theorem 7(1) applied to $\Gamma; [\Gamma]_1^{I_0} \vdash P$ and $\Gamma; [\Gamma]_1^{I_0} \vdash_1 P \xrightarrow{(\Gamma')u!(V)} Q'$, we derive $\Gamma + [\Gamma]_1^{I_0} \vdash u : S$, $\Gamma + \Gamma' \vdash V : T$ and $S \lesssim \langle T \rangle^0$. By Theorem 7(2) applied to $\Gamma; [\Gamma]_1^{I_0} \vdash P$ and $\Gamma; [\Gamma]_1^{I_0} \vdash_1 P \xrightarrow{u?(F)} Q''$, we derive $\Gamma + [\Gamma]_1^{I_0} \vdash u : S$ and $S \lesssim \langle \text{schof}(F) \rangle^I$. Since Γ is channeled, $S = \langle S' \rangle^\kappa$, for some S' , κ . Therefore, by contravariance of $\langle \cdot \rangle^0$, covariance of $\langle \cdot \rangle^I$, and transitivity of \lesssim : $T \lesssim \text{schof}(F)$. The statement $\Gamma + \Gamma' \vdash V \in F \rightsquigarrow \sigma$ follows directly from Proposition 8 because Γ is channeled.

Next, if $\Gamma \vdash_1 P \xrightarrow{(\Gamma')u!(V)} Q'$ and $\Gamma \vdash_1 P \xrightarrow{u?(F)} Q''$, by definition of the transition relation, $P = C[\text{spawn } \{P'\} P'']$, where $C[]$ is a context, and $\Gamma + \Gamma''' \vdash_1 P' \xrightarrow{(\Gamma'')u!(V)} Q'_1$ and $\Gamma + \Gamma''' \vdash_1 P'' \xrightarrow{u?(F)} Q''_2$, where $\Gamma' = \Gamma'' + \Gamma'''$. Progress holds for $\text{spawn } \{P'\} P''$ because every premise of (TR8) is true (the constraint $\text{dom}(\Gamma') \cap \text{fn}(P'') = \emptyset$ may be easily enforced by alpha-conversion). This result may be lifted to P by means of rules (TR3), (TR6), (TR7). \square